THESIS

A STUDY ON MODELING APPROACHES IN DISCRETE EVENT SIMULATION USING DESIGN PATTERNS

by

Kim Leng Koh

December 2007

Thesis Advisor:	Arnold H. Buss
Second Reader:	Man-Tak Shing

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** December 2007	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE A Study on Modeling Approaches in Discrete Event Simulation Using Design Patterns			**5. FUNDING NUMBERS**
6. AUTHOR(S) Kim Leng Koh			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) Singapore Technologies Electronics (Training & Simulation System) Pte Ltd 24 Ang Mo Kio St 65 Singapore 569061			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

Many discrete event simulation (DES) systems have been built using Simkit as the underlying infrastructure. Simkit advocates a modeling paradigm where DES applications can be rapidly built with simple, independent modules that are assembled in a component-like fashion. This modeling paradigm encompasses several modeling approaches—active role of events, entities as independent components, and chaining components to enable interactivity—that are excellent ways of building a DES system.

This thesis is inspired by the great work achieved in the mechanisms of the underlying infrastructure. Detailed study of the enabling mechanisms and design patterns was conducted. Design patterns are proven design solutions that embody best practices of software-design concepts; this thesis proposes new design that incorporates suitably identified design patterns into the mechanisms of the infrastructure to bring out the elegance of design, robustness, and maintainability that heighten the maturity of a simulation engine.

The result of this research work has been a success; several design patterns have been identified and incorporated into a new design of the mechanisms behind a simulation engine. A DES application that was built for the SEAs project was able to switch over to run on the new simulation engine while keeping its business model intact.

14. SUBJECT TERMS Discrete Event Simulation, Event Graph Methodology, Simkit, Design Patterns, UML, Modeling Paradigm, Java, Object Oriented, Framework, Infrastructure, Simulation Engine, Component Listener, LEGO, Force Protection & Port Security (FPPS)			**15. NUMBER OF PAGES** 115
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

A STUDY ON MODELING APPROACHES IN DISCRETE EVENT SIMULATION USING DESIGN PATTERNS

Kim Leng Koh
Civilian, Department of Defense of Singapore, Defense Industry
B.A.Sc., School of Computer Engineering, Nanyang Technological University, 1998

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN MODELING, VIRTUAL ENVIRONMENTS, AND SIMULATION (MOVES)

from the

NAVAL POSTGRADUATE SCHOOL
December 2007

Author:	Kim Leng Koh
Approved by:	Professor Arnold H. Buss Thesis Advisor
	Professor Man-Tak Shing Second Reader
	Professor Rudy Darken Chairman, Department of MOVES

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Many discrete event simulation (DES) systems have been built using Simkit as the underlying infrastructure. Simkit advocates a modeling paradigm where DES applications can be rapidly built with simple, independent modules that are assembled in a component-like fashion. This modeling paradigm encompasses several modeling approaches—active role of events, entities as independent components, and chaining components to enable interactivity—that are excellent ways of building a DES system.

This thesis is inspired by the great work achieved in the mechanisms of the underlying infrastructure. Detailed study of the enabling mechanisms and design patterns was conducted. Design patterns are proven design solutions that embody best practices of software-design concepts; this thesis proposes new design that incorporates suitably identified design patterns into the mechanisms of the infrastructure to bring out the elegance of design, robustness, and maintainability that heighten the maturity of a simulation engine.

The result of this research work has been a success; several design patterns have been identified and incorporated into a new design of the mechanisms behind a simulation engine. A DES application that was built for the SEAs project was able to switch over to run on the new simulation engine while keeping its business model intact.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

The author would like to express deep gratitude and sincere appreciation to his thesis advisor, Professor Arnold H. Buss, for his patience, guidance, and encouragement as the author worked on this thesis. His magnanimity, as the creator of Simkit, to give the author the opportunity to endeavor into this research work in the pursuit of maturity and elegance of software design is indeed valuable and extraordinary. The author always enjoyed a gratifying experience during those weekly discussions on numerous aspects in software engineering and modeling and simulation.

The author would also like to express sincere appreciation to his second reader, Professor Shing Man-tak, for all his guidance in design patterns. He has provided relentless effort to the author to better grasp the profound concepts of design patterns. His guidance has been extremely valuable and helpful to the author in this research work.

The author would like to thank Miss Margaret Davis for her editing, which assisted the professional and succinct expression of key concepts.

The author would like to express gratitude to Singapore Technologies Electronics (Training & Simulation System) for the sponsorship that gave the author the opportunity to take up this academic pursuit.

Last but not least, the author would like to thank his classmate Mr. Boon Leng (Ryan) Tan for his encouragement during the ups and downs of this academic pursuit. The author just wants to express appreciation to him for his presence all the while.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. OVERVIEW

The software industry has probably seen the most fascinating evolution in modern human history. Historically, the software application was properly handcrafted by a computer scientist using punch cards and queued on a huge mainframe, back in the sixties. The first software application the author built was an assembly-language subroutine that was submitted as a job to a DEC digital-computer mainframe. Today, a software application can easily be generated automatically on the fly through code generators and run instantaneously once it has been designed, all on a small laptop. In a matter of decades, the advent of technology has enabled software applications to seep into our daily life, rendering them indispensable. Software development has matured rapidly in diversity. Nevertheless, the satisfaction in getting a new software design to run on the laptop never failed to give the software developer a sense of fulfilling joy, just as the author was thrilled when that assembly subroutine worked. The computer scientist would not be any less elated when that first software application that was handcrafted on punch cards crunched successfully.

The first event-driven simulation application that the author built is the arrival-process. It is a simple, self-propelling module that exposes the modeling approaches— in modeling events, entities, and components—that the Simkit modeling paradigm advocates in building a DES system. It portrayed an abstract aspect of a system. Such an abstracted view of a system has great implications, as it leads to endless possibilities of how a system could be spawned out of this arrival-process. The behavior captured in this independent module soon finds itself highly reusable and maintainable, as this abstractedness is commonly found across most systems. The event-driven perspective of system analysis and the software-development approach encourages simple and independent software modules that are loosely coupled to be developed. They become easily reusable and an elegant way of building a complete system simply through assembling of modules. These are essential qualities in building systems in an object-oriented fashion.

This approach of building simulation applications rapidly is made possible through an underlying simulation engine infrastructure. The modeling paradigm of the simulation engine framework determines how independent a software module can be built. There must be mechanisms of the simulation engine at play to support the abstracted view of an independent module to run in a self-propelling fashion. There must be other encompassing mechanisms working hand in hand to facilitate the assembly of many independent modules to construct a complete system.

While the abstracted behavior of the arrival process is highly reusable, a robust simulation engine is also highly, if not more, reusable and deployable, as the mechanisms continue to facilitate new applications to be built rapidly across different domains. As one imagines, the software engineering involved in building the simulation engine is more abstract and, at times, daunting. However, the satisfaction and challenges in engineering a robust and elegant simulation-engine infrastructure that can bring pleasure to application developers when their applications run is even more joyful, thrilling, and motivating.

B. MOTIVATION

Numerous research works have attempted to conquer the problems of specific domains. Their challenges essentially motivate them to dwell on how DES simulation can be suitably applied.

The nature of this research work, however, revolves around framework mechanisms and the interworking of these simulation-engine mechanisms in realizing the modeling paradigm. One challenge that motivates this work is the relentless drive for elegance in the design of these mechanisms. Elegance in the design of a piece of software reflects the quality of engineering workmanship just like a piece of art. The appreciation and pursuit of elegance in the way mechanisms are created is a motivating challenge. In a robust simulation engine, there is an inevitable tussle between elegance and performance. While performance holds the key that enables proliferation of the simulation engine, elegance holds the key of flexibility, extensibility, and maintainability. The motivation in this research work is to reexamine existing design and provide insights on how conflicts between performance and elegance may be averted.

Unlike other research that broadens the horizon of DES across different domains, this thesis delves into the robustness of the mechanism design in the underlying simulation engine framework, which eventually compliments other research works that broaden the applicability of DES. This research attempts to study new design concepts that heighten the maturity of a DES simulation-engine framework infrastructure.

C. OBJECTIVES

One avenue where the element of elegance can be injected into system design is to incorporate suitably identified design patterns. Design patterns (DPs) are not new designs, but proven solutions that have evolved over time. They are particularly suitable in mechanism-oriented framework systems.

The objective of this research work is to study the behavioral characteristics of design patterns. Relevant design patterns that are applicable to the simulation-engine framework will be identified. A new design that attempts to suitably incorporate design patterns into the simulation-engine framework will be proposed.

A well-structured object-oriented architecture is full of design patterns [1]. This is one way by which the quality of a system is judged [1]. The elegance is reflected in the way suitably identified design patterns that have been applied, or possibly created, produce elegance through simplicity of design, flexibility and modularity in components that made up the architecture, and a high level of reusability for the components that constitute the architecture.

D. THESIS ORGANIZATION

Chapter II conducts literature review on the conceptual paradigm, methodology, modeling language, and technology that this research is based on. Chapters III and IV present the detailed understanding on the behavioral characteristics of design patterns that are relevant and applicable in this work. Chapter V looks at key features of the Java framework that have concepts similar to the mechanisms of the simulation-engine framework. Analysis of these key features reveals the existence of behavioral characteristics of DPs. Chapter VI studies some key modules and mechanisms of the current Simkit simulation-engine framework. The analysis of the behavior established resemblances of design pattern. Chapter VII discusses a proposed design that incorporates design patterns for those key modules and mechanisms, and empirical tests and compatibility of the new design are carried out. Chapter VIII summarizes the research carried out in this thesis.

II. BACKGROUND

This chapter focuses on the key elements of the discrete-event paradigm and the basic concepts of event-graph methodology. They form the theoretical background of Simkit and are the fundamentals of this thesis work. This is followed by a discussion of the features of Simkit that this thesis will be looking at. A discussion of the key features of UML modeling language is conducted, as this work will be using UML extensively in all analysis and design. Finally, a brief description of design patterns as used in this research is presented.

A. DISCRETE EVENT PARADIGM

Discrete-event simulation describes the modeling of a system over time, where the system-state variables change instantaneously at separate points in time [2]. These specific points are where events occur. An event, specifically a discrete event, is defined as an instantaneous occurrence in the discrete-event model that may change the state of the system [2]. Discrete-event models have state trajectories that are piecewise constant [3]. These discrete events are the points in time when at least one system-state variable changes its value [3]. During an event, simulation time stays constant and unchanged, unlike the continuous tickling of real-time clock. In discrete-event simulation, the simulation time is an indicator of the occurrence of events. This is the fundamental concept on which a discrete event in a discrete-event world is built, and it leads to viewing the simulation world entirely from the event’s perspective. This event-oriented perspective, combined with the discreteness of events, has resulted in the concept of the discrete-event paradigm within the field of modeling and simulation.

While the discrete-event paradigm is event oriented, it is still the modeling of a system over time, particularly simulation time. Only after an event has occurred is simulation time updated to the scheduled time in which an event occurred. As such, the simulation time of a system advances between the occurrences of events [3]. The idea of time advancement, specifically simulation-time advancement, is a vital aspect of the system. The simulation’s clock is defined as the variable in a simulation model that gives the current value of simulation time [2]. It is important to be aware that simulation time is unrelated to the computational time needed to run a simulation model [2]. The simulation's clock is updated with the simulation time of occurrence of the discrete event. In practice, the unit of time for the simulation clock is never stated explicitly in a model written in the programming language [2].

There are basically two approaches to simulation time advancement. One approach looks at the advancement of the simulation's clock according to the time of the next executing event [2]. Where there are sparse events in a system or when there is a vast difference in the simulation time of occurrence of events, the simulation time will make large advancements as events occur. The other approach is a fixed-incremental advancement of simulation time [2]. A simulation's time is advanced periodically at fixed interval. Events with a big difference in time of occurrence may need to wait for the periodic-time advancement to elapse.

Both approaches are being adopted in the implementation of discrete-event simulation systems. In fact, the latter concept of advancing a fixed increment of simulation time is a special case of the former concept [2]. There could be a repetitive occurrence of events at fixed or periodic time intervals. As such, the modeling of time advancement based on the occurrence of each next executing event is the more generic and versatile approach.

B. EVENT GRAPH METHODOLOGY

In discrete-event simulation and modeling there are three system-structuring approaches [4] or worldviews: activity scanning, process interaction, and event scheduling. Graphical representations like block diagrams [5], process networks [6], activity wheel charts [7], activity lifecycle diagrams [8], and Petri-net diagrams [9] have facilitated the proliferation of activity-scanning and process-interactive worldviews, which led in turn to the popularity of process modeling and activity modeling as the more common approaches in system analysis. To a certain extent, these two conventional approaches have been used to provide analysis of event-driven systems, even though their focus revolves around entities and attributes. Unfortunately, the true abstractedness of event analysis in event-oriented systems easily loses focus, defeating the purpose of system analysis. The key to facilitating qualitative system analysis is both correct recognition of the nature of the system and an apt approach to analysis.

The event graph, as advocated by Schruben [4] is an attempt to establish a graphical technique for visualizing event-oriented system structures. This graphical representation is simple in nature and its expression strongly reflects the event-driven nature of event-oriented systems. The strength of its simplicity has tremendous value in enabling ease of analysis, especially in perceiving the sophistication of event-scheduling approaches in discrete-event system simulation [4]. The focus of analysis using event graphs revolves around the notions of system events, system-state variables influenced by the occurrence of system events, events that determine future events, and events that cancel future events.

In event-graph notation, an event that results in the change of a system-state variable is represented as a vertex (single node). The relationship between two events is represented by a directed edge (single arc). A directed edge contains informative notation that indicates which event schedules or triggers the occurrence of another event, when the scheduled or triggered event will occur, and the conditions that bring on the scheduled or triggered event.

Figure 1 illustrates a simple scheduling of events. There are two vertices, event j and event k. Event k is scheduled to occur after t time units have elapsed following the occurrence of event j. This is provided that condition i is fulfilled at the point when event j has completed its execution.

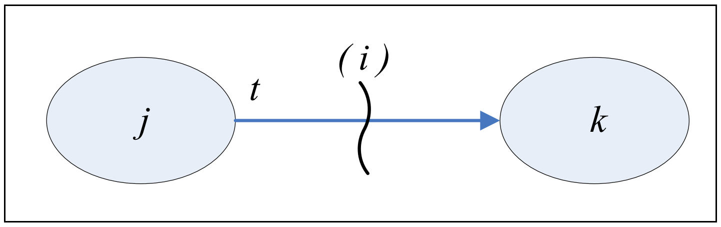


Figure 1 Simple Scheduling of an Event (From [4])

The strength of event graphs as a graphical representation lies in the simplicity with which they enable a direct focus on the analysis of the set of system events, the relationship between these events, when an event will be scheduled, and the condition that materializes the relationship. An event may schedule or trigger several other events. If so, there will be several arcs out of event *j* to several other event vertices. An event can be instantaneously scheduled, as illustrated in Figure 2. In this case, the time-unit notation will be completely omitted, but there could still be condition *i*, as illustrated in Figure 2 that must be fulfilled for event *j* to be scheduled.

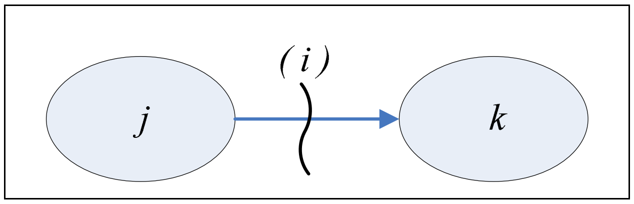


Figure 2 Instantaneous Scheduling of an Event

An event can also be unconditionally scheduled, as illustrated in Figure 3. In this case, the conditional notation will be completely omitted but there could still be time-unit *t*, as illustrated in Figure 3 that needs to elapse for event *k* to be scheduled.

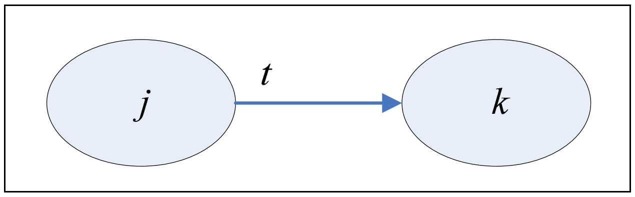


Figure 3 Unconditional Scheduling of an Event

An event can be unconditionally and instantaneously scheduled, as illustrated in Figure 4. In this case, both the time-unit notation and conditional notation will be omitted.

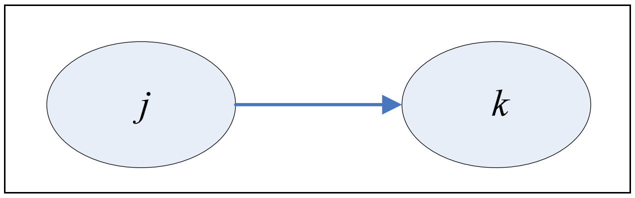


Figure 4 Unconditional, Instantaneous Scheduling of an Event

An event can schedule itself, as shown in Figure 5. In this case, after *t* time unit has elapsed and condition *i* is fulfilled, event *j* will be scheduled to execute again.

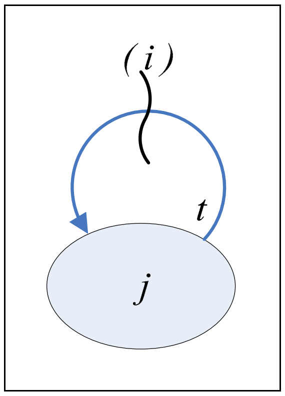


Figure 5 Self-Scheduling of an Event

In event-graph methodology, while an event can be scheduled, it can also be cancelled. Figure 6 presents an event-graph representation showing cancellation of events. In this case, event *k* will be cancelled after *t* time unit has elapsed following the completion of event *j*, provided that condition *i* is fulfilled when event *j* has completed execution. The dotted scheduling arc indicates the cancellation of an event.

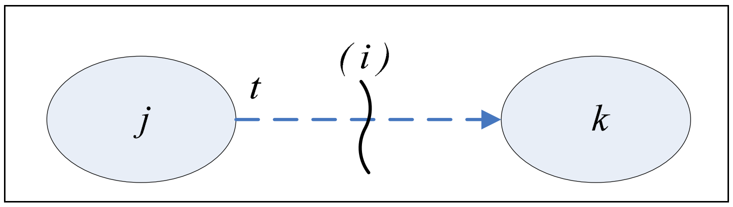


Figure 6 Cancellation of an Event (From: [4])

The graphical representation of the system using event-graph methodology cannot be misinterpreted as the program-flow chart of the system. An event graph is a representation of system structure that will be used as a preliminary step in top-down simulation-model development [4]. The graphical notation of an event graph is simple, yet contains enough information for system analysis. Event-graph methodology's representations provide a worldview that facilitates the analysis of event-driven systems with the abstractedness of events totally unveiled.

C. SIMKIT

Simkit is the discrete-event simulation (DES) engine created by Buss [10] at the Naval Postgraduate School. Without a simulation engine in place, an application must cater specifically and individually to the design and implementation of when and how a model computes—issues that in every application require resolution. However, having each application cater to when models compute is reinventing the wheel, because under the DES paradigm, they all implement the same conceptual approach. An application that addresses how models compute is in fact the focus in developing a unique solution fulfilling the requirements an application is built for. A more sensible approach in building applications in the domain of DES is to identify a robust simulation engine to address when models will compute, while the models focus on how they will compute, so as to address the requirement of the problem space. With an underlying simulation engine that takes care of the core organization and triggering mechanisms, simulation-application development by model developers is able to concentrate on software modeling of the physics of their domain.

A robust simulation engine assumes the role of designing and implementing the core mechanisms needed to associate all application models in a generic fashion, chaining these models altogether, identifying each of them according to their priorities without discriminating or distinguishing any specific model, scheduling them, and finally triggering them to compute in an orderly and efficient manner. Simkit is one robust simulation engine developed for building simulation applications in DES.

Simkit is written entirely in the Java programming language. Simkit consists of a suite of Java libraries that constitute the discrete-event simulation-engine framework, where discrete-event models can be written and developed. Unlike some other simulation engines, in which the simulation engine and application models are programmed in different languages, Simkit-based applications are developed in the same language that the simulation engine is written in. Application-event modeling codes are also written in Java, enabling Simkit and Simkit-based applications to be platform independent. Any operating system that allows Java Virtual Machine to reside on the host machine will be able to support Simkit and its applications. The platform independence of Simkit-based applications is tied to the availability of Java Virtual Machine on the host machine. Over the years, Simkit has developed a rich set of application-programmer interfaces (APIs) for models to interact with. Simkit's simulation-engine framework provides several straightforward mechanisms to allow newly developed application models to be chained generically to run as a single discrete-event simulation executable.

Simkit as a DES simulation engine embraces the event graph as the underlying methodology [10], and all the concepts of this methodology have been implemented in Simkit. In addition, through the versatility of event graph, Simkit has extended this methodology to include additional annotation, augmenting graphical representations to include richer information in event scheduling. The following paragraphs will briefly highlight the extensions to event graph that have been incorporated in Simkit.

Event graph methodology has no restrictions on the number of events that can be scheduled. In fact, several events can be scheduled simultaneously. In practice, when simultaneous events occur, it makes sense to incorporate the notion of priority; events with higher priority should occur before other scheduled events. The event graph can therefore be extended to include notation of priority levels, which are depicted within a circle along the scheduling edge, towards the tail of the scheduling arc, as illustrated in Figure 7 where one of the events needs a priority annotation. In this case, once event *h* has completed its execution, three events—*i*, *j*, and *k*—are scheduled simultaneously as unconditional, instantaneous events. The annotation of *P1* along the scheduling edge of event *j* indicates that *j* has a higher priority than event *i* and *k*, which have default priority. Simkit ensures that *j* will execute before *i* and *k*.

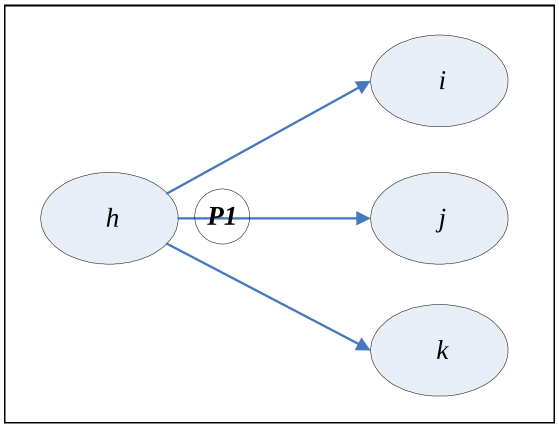


Figure 7 Priority Scheduling of Events

In the development of an application, inevitably the ability to perform some form of data passing among events must be present (that is, beyond the means to schedule events at all). This is analogous to the ability to make function calls and the inevitable need to pass data into the function call as function parameters. Basic event-graph representation reflects the scheduling of events without any representation of data passing. This lack has been rectified in Simkit by means of data-passing notation, with the data encased in a square along the scheduling edge, towards the tail of the scheduling arc [3]. The scheduled event will reflect a corresponding match of the data with the data encased in brackets [3]. This is illustrated in Figure 8 with data being passed in scheduling another event. In this case, after event *j* has completed it execution and condition *i* is fulfilled, event *k* will be scheduled after *t* time units have elapsed. Data *q* in event *j* will be set to values of data *p* [3].

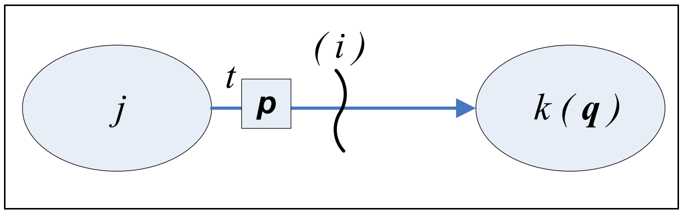


Figure 8 Data Passing in Scheduling Events

Event-graph methodology has included the concept of event cancellation, but Simkit adds detailed interpretation that works smoothly with extensions and with scheduled events that include data passing [3]. Figure 9 shows such a cancellation. In this case, after event *j* has completed execution and condition *i* is fulfilled, the first occurrence of event *k* whose values of data *q* matches data *p* will be removed from the system. If no such instance of event *k* can be found, nothing is removed from the system and nothing happens [3]. In the cancellation of events, data *p* is optional. When there is no data *p*, Simkit identifies the first occurrence of event *k* with no argument and removes it from the system. If no such instance of event *k* can be found, nothing is removed and nothing happens. This interpretation of event cancellation is an extension of the event-graph methodology by which Simkit brings a finer level of detail to the concept.

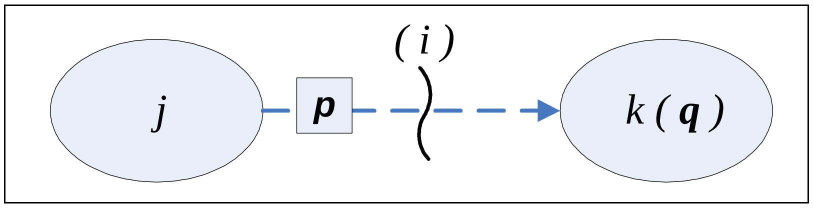


Figure 9 Simkit: Cancellation of an Event

The Simkit DES framework is an implementation of the LEGO [11] framework, and as such supports the key concepts of component-based simulation modeling [12]. Simkit emphasizes several of these concepts in steering its modeling paradigm towards component-based simulation modeling. One key is the definition of a single component as a basic, monolithic programming entity [10] that fully encapsulates an independent set of event-graph logic [11]. The mechanism of associating components in Simkit underscores that linking these components together is a rapid yet robust approach in building larger, more complex systems [11]. The enabling mechanism of Simkit ensures the possibility of loose coupling [10] and substantial reusability among components. Simkit's enabling mechanism relies heavily on establishment of a common interface among components. The following paragraphs will highlight some key graphical representations that Simkit incorporates to steer its modeling paradigm in the direction of component-based simulation modeling.

Each component in Simkit has an independent set of event-graph logic. The triggering of events within a component can cause dependency by other components, in such a way that a system event occurring in a source component triggers the execution of the same kind of event in another dependent or listening component. This is the underlying concept of the listener mechanism, in which there is an event-source component, an event-listener component, and a line that connects the two with a stethoscope-like [12] symbol on the source end. In Figure 10 component A is the event-source component and component B is the event-listener component. An event in *A* will trigger a similar kind of event in *B*. The association of these loosely coupled components allows the dependent listener, *B*, to sense (listen to) the triggering of events from source-component *A*. Simkit does not limit the number of listeners that can tap a source component, or, contrariwise, limit the number of sources a listener can listen to. Nor is there anything to prevent a component from adopting a dual role as both source and listener.

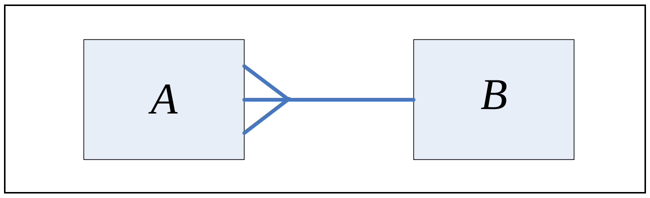


Figure 10 Component-Listener Mechanism

For two components to exploit the listener mechanism, strict conformity with the exact same kind of system event must be observed by both source and listener. One way to support the triggering of a specific event from an event-source component on another specific event in the event-listener component, while ensuring the integrity of both, would be a mechanism that explicitly wraps the events. This is the concept underlying the adapter mechanism, in which there is an event from the source component, an event from the listener, and a double line that connects the two with a stethoscope-like near the source-component end. This is illustrated in Figure 11 where *c* is a specific event from event-source *A* and *d* is a specific event from event-listener *B*. Occurrence of *c* will trigger event *d* only.

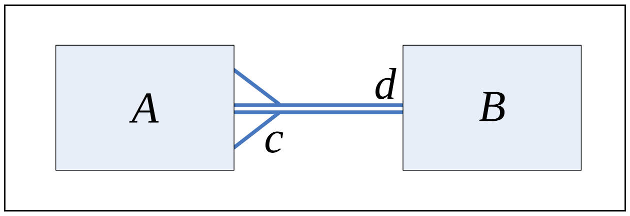


Figure 11 Component Adapter Mechanism

Application-model developers using Simkit must know Java, basic concepts in event-graph methodology, and Simkit's event-graph extensions. Simkit's simulation engine is in fact an embodiment of the DES paradigm, with an intricate design that emphasizes clean dependency and considerable loose coupling between simulation-engine libraries and application models. Development can be carried out independent of any enhancement that needs to be implemented in the simulation-engine framework.

D. UML

The early 1990s saw a surge of interest in the object paradigm, and related technologies proliferated. It was also a time when new object-oriented (OO) languages were created, such as SmallTalk, Eiffel, C++ and Java. Many object-oriented design methodologies and graphical representations emerged, each making its own ingenious attempt to embrace and represent the same fundamentals of the OO paradigm [13]. The abundance of these differing notations unfortunately led to much confusion and concern about how adoption of the object-oriented paradigm by software developers might be impeded as a result. The need for a unified notation was critical.

In 1997, the Unified Modeling Language (UML) was first formalized as UML 1.0. Proposed initially by Grady Booch, Ivan Jacobson, and Jim Rumbaugh [13] of Rationale Software, UML was a collaborative effort among top industry leaders to consolidate the best features of various OO approaches in a vendor-independent, general-purpose modeling language [14] and notation [13]. Since then, UML has been the de-facto standard in various domains of the software industry and was adopted by Object Management Group (OMG) as a bona-fide industry standard [14] [15] [16]. UML 2.0 [14] [15] is the current release.

As the standardized notation in software modeling, UML has raised awareness of the value of modeling in dealing with software complexity [13]. UML is a suite of notations that attempt to specify, visualize, and document models of software systems, including structure and design [17], to represent requirements, relationships, and other developmental concepts in software analysis and design, such that the software better represents the system modeled. One benefit that UML offers is a common basis for understanding and communication among analysts, designers, and coders throughout the process of software building, so that ideas can be better conveyed, ambiguities better clarified, feasibility better gauged, and contractual delivery timelines better managed.

UML 2.0 has been revised to address web-based applications and service-oriented architectures and to account for the analysis, design, and development phases of large-scale software systems. UML 2.0 has emerged as a standard for model-driven development, which emphasizes models as the primary artifacts of software design [13], leaving code generation mostly to abstracting and automating technologies.

UML has several graphical-representation notations or diagrams that constitute the key features of this modeling language. These diagrams fall into two broad categories: structural modeling of the system and behavior modeling. Diagrams that belong to structural modeling of systems are class, package, object, component, and deployment diagrams [18]. Diagrams belonging to the behavioral modeling of systems are use-case, sequence, collaboration, state-chart and activity.

A *class* has been defined to describe a set of objects that share the same set of specifications, constraints, and semantics [15]. Class diagrams give a static overview of the system as they illustrate, all at a glance, relationships among classes [18]. Class diagrams capture essentially three relational dynamics—association, aggregation, and generalization—among classes in a system. Figure 12 illustrates that *Base* class is a generalization of *Derived* class; *Derived* class aggregates *Part* class; and *Derived* class is associated with *Person* class. The numbers indicate the multiplicity and cardinality of instances of the relationship.

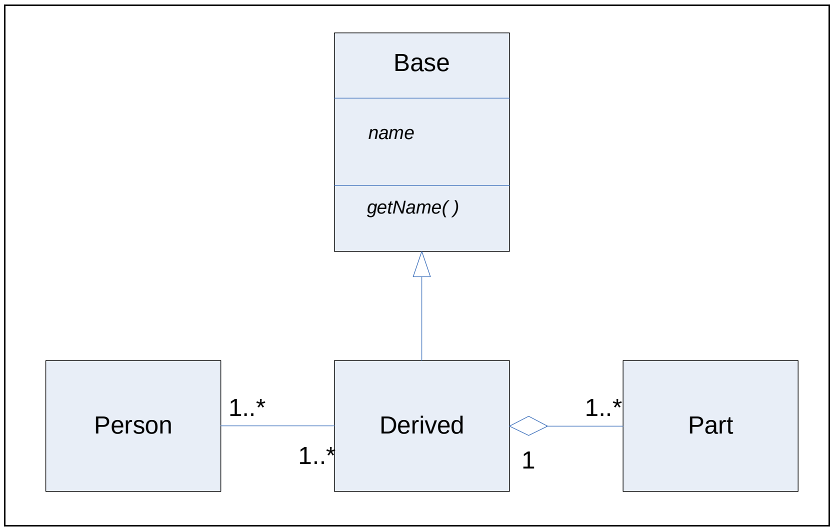


Figure 12 Class diagram

A *package* is a collection of logically related UML elements [18] and optionally provides a namespace for this group of elements [15]. It is common practice to group related classes into a common package. Package diagrams capture dependencies among packages [18]. Figure 13 illustrates a DB Package that is dependent on a transaction package.

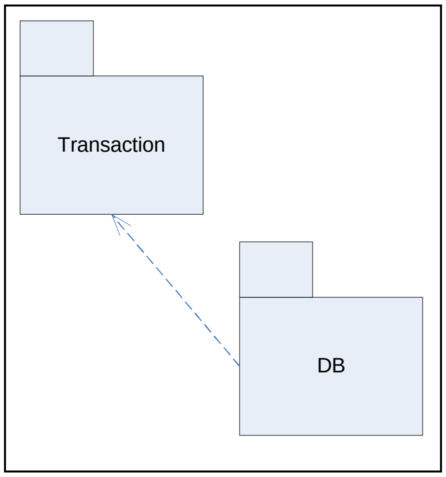


Figure 13 Package diagram

An object *diagram* shows the instances of the classes that each belongs to. This is useful in illustrating complicated relationships, e.g., recursive relationships among object instances of a class [18]. Figure 14 illustrates an object instance named *John* that belongs to the *Person* class.

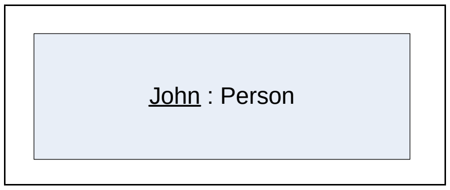


Figure 14 Object Diagram

A *component* is defined as a code module; a component diagram shows the physical analogs of a class diagram [18]. A deployment diagram, the last element in structural modeling of a system, shows the physical configurations of software and hardware. Figure 15 illustrates two nodes: server and client. The *Buyer* component, deployed on a client node, interfaces with the *Seller* component, deployed on a server node.

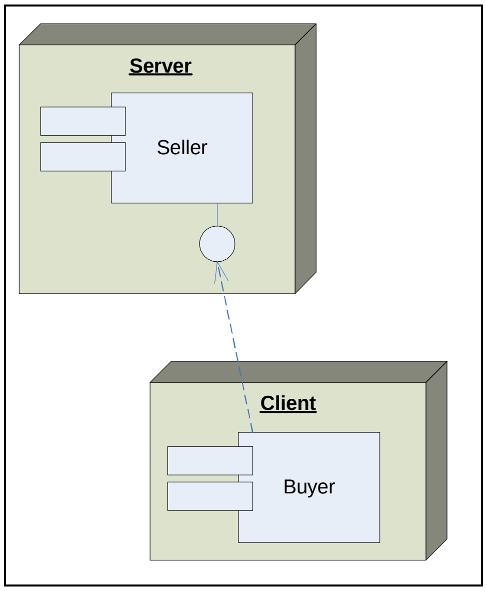


Figure 15 Deployment Diagram of Components

A use-case diagram describes the system from an external observer's point of view. The modeling approach in using use-case diagrams is to capture what capabilities the system has while intentionally ignoring how the system gets those capabilities implemented [18]. It is closely connected to scenarios and the actors enacting in the scenarios. Figure 16 shows an *Operator* actor involved in *New Reservation*, *Modify Reservation*, and *Delete Reservation* use cases.

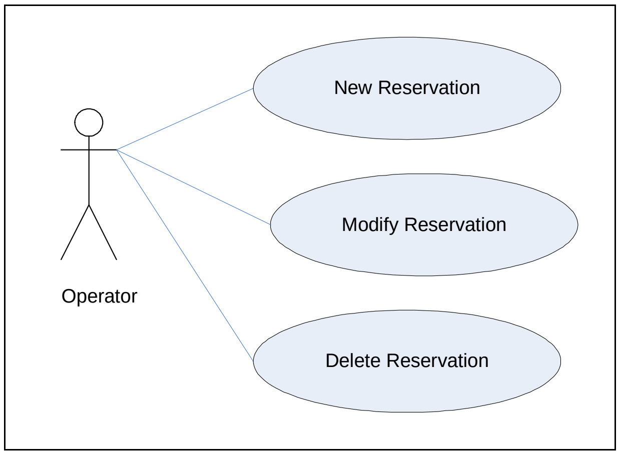


Figure 16 A Use-Case Diagram

A sequence diagram is an interaction diagram that illustrates the operations that will take place among objects or classes [18]. There is a notion of the passage of time as it illustrates the sequence of interaction among these elements. In Figure 17 the *Person* class incites the *Record-Manager* class to create a new record. The *Record-Manager* class then creates a record in a sequential manner. The collaboration diagram is another form of interaction diagram [18]. The sequence flow in collaboration diagrams, however, focuses on the role of objects. There is no restriction to ensure sequential flow of time in sequences that link objects.

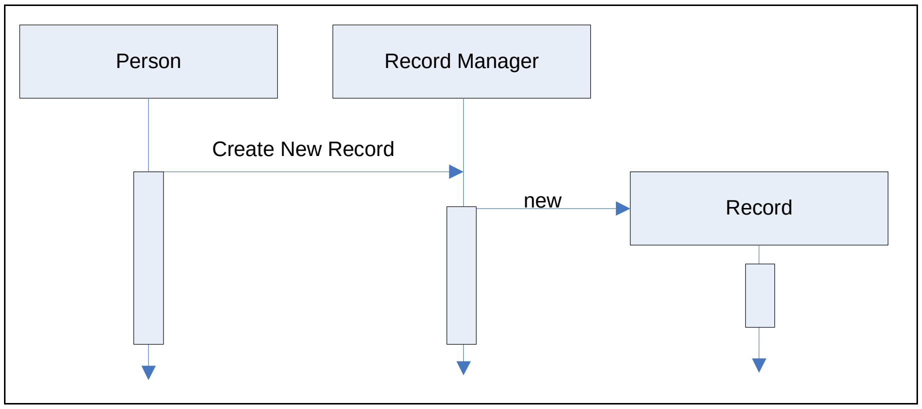


Figure 17 Sequence Diagram

UML 2.0 is now a standard for rich modeling features organized as a language-modeling architecture [13]. Its versatility has led to modularity and a gradual approach to adoption. It encourages the user to learn and apply a suitable subset of UML modeling features that best support a problem domain, rather than to master the full extent of the language. As the experience and knowledge of both the system and modeling language grow, rich new capabilities can be harnessed to express intricacies. UML has been applied widely in many domains, including direct modeling of software architecture, complex system interactions, flow-based application models, business processes, and system engineering [13]. The flexibility of the language has seen its applicability across many platforms [14], ranging from small, individual software modules to large, complex software systems of systems.

E. DESIGN PATTERNS

In computer science, particularly in software engineering, the idea of design patterns took off in the late 1990s and was ubiquitously applied across multiple industrial domains. Perhaps only a few remember that design patterns actually originated with the collecting of architectural concepts pursued by the American architect Christopher Alexander, whose field was civilian architecture [19].

Gamma *et al* [1] define design patterns as simple and elegant solutions to problems in object-oriented software design [1]. Design patterns are, in fact, not newly crafted designs for new problems. They are proven solutions that evolved out of programming pain and success in the many systems that shaped their existence. A design pattern can be considered a general, repeatable solution that can be applied to the recognizable, repeatable problems that occur in every new problem space. Design patterns are created to record instances of good design in object-oriented software, so they can be reapplied rather than rediscovered. They were created in the expectation that good design and successful architecture are recyclable.

Christopher Alexander describes design patterns as follows:

> Each pattern describes a problem which occurs over and over again in our environment and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice [19].

A design pattern is not the first few designs created as a software solution. Design patterns are a slow outcome, as developers explore rounds and rounds of redesign, striving to achieve higher reusability and flexibility. The ability of a solution to be applicable time and again in new-yet-familiar situations is what makes a solution a design pattern. A design pattern is therefore not a piece of code. One might say that a design pattern captures the gist of a solution [1]. Applying apt patterns to problems truly transforms object-oriented designs, making them more flexible [1], simple, elegant, and optimally reusable. At the analysis phase, applying relevant design patterns can help in choosing design alternatives that make a system reusable and avoiding alternatives that compromise reusability [1]. To a certain extent, design patterns may even improve the documentation of software systems, and thus be helpful in maintaining existing systems.

Design patterns are neither specific algorithm designs like a FIFO queue, nor a linked list, nor complex, domain-specific designs for an application. Design patterns are descriptions of communicating objects and classes [1] that portray a generally reusable solution to a design problem. A design pattern abstracts and identifies the key aspects of a common design structure and makes it useful for creating a reusable object-oriented design [1].

The many design patterns that have been documented by the "Gang of Four" [1] fall broadly into three categories: creational, structural, and behavioral, according to their purpose. It is no surprise that design patterns are applicable as useful solutions for different problems among components of the Simkit simulation engine. Simkit's simulation engine is an object-oriented architecture implemented as a DES framework that supports the rapid development of DES simulation applications. The introduction of design patterns into Simkit design would heighten the maturity of its architecture as a DES framework, elevate its elegance in terms of mechanistic simplicity and maintainability, and propel the flexibility and reusability of the various core components towards optimal reusability.

III. CREATIONAL DESIGN PATTERN

This chapter conducts detailed discussion of two design patterns that belong to the creational category. The approach is to focus on the situation that each design pattern arises, its applicability and its design structure, to illustrate its characteristics. These two design patterns have been suitably applied in the design work of this thesis research.

A. FACTORY DESIGN PATTERN

1. Situation

In building a software application in an object-oriented fashion from scratch, every class instance (object) is instantiated from the respective concrete class that is being designed for the application. In building a software application using a (software) framework, it is still the responsibility of the framework to instantiate every object from the required concrete class. However, a framework would not be aware of the newly designed concrete classes that a new application needs, because the framework was created a priori. The framework knows, nevertheless, when an object of its respective concrete class that the application needs must be instantiated and manages the relationship among these objects, while unaware of what concrete classes will ever be written. For a framework, a dilemma [1] exists where an unknown specific concrete class needs to be contacted to instantiate an object. One way to solve this dilemma would be a means to encapsulate the knowledge of which specific concrete class is needed to instantiate the object and move this knowledge out of the framework [1]. The need to unveil this encapsulated knowledge would be delayed till the point when the object needs to be instantiated. This is the situation where the factory design pattern was meant to provide a solution to the dilemma that the framework encounters.

2. Intent

The intent of using a factory design pattern is to define an interface—the factory method—that instantiates an object. The factory provides the means to defer as it delegates the subclasses—which encapsulate the required knowledge away from the framework—to decide which specific concrete class actually instantiates an object [1]. This design pattern is applicable in situations where there is no means to anticipate the specific concrete class to instantiate objects beforehand. There arises a need to localize and delegate this responsibility to subclasses that will be able to carry out the task duly at runtime. It is also applicable when subclasses are the concrete classes that can instantiate object while an abstract class could not. It is useful when the subclasses are required to specify which specific object to instantiate.

3. Design Structure

The basic design of classes of the factory design pattern is illustrated in Figure 18. The *BaseClass* and *Client* classes are abstract classes of the framework. *BaseClass* is the factory class. The framework defined the means whereby *Client* would contact *BaseClass* when an object needs to be created, as illustrated by the dotted arrow that points into *BaseClass*. *CreateObjectOfRequiredClass* is the factory method—the interface—that encapsulates the required knowledge away from the framework. The subclasses, *AClass* and *BClass*, are the concrete classes that will create each of their respective objects. In this design, sub-classifying provides the means to contact specific concrete subclasses to create an object when the delayed and delegated action of creating an object needs to take place at runtime.

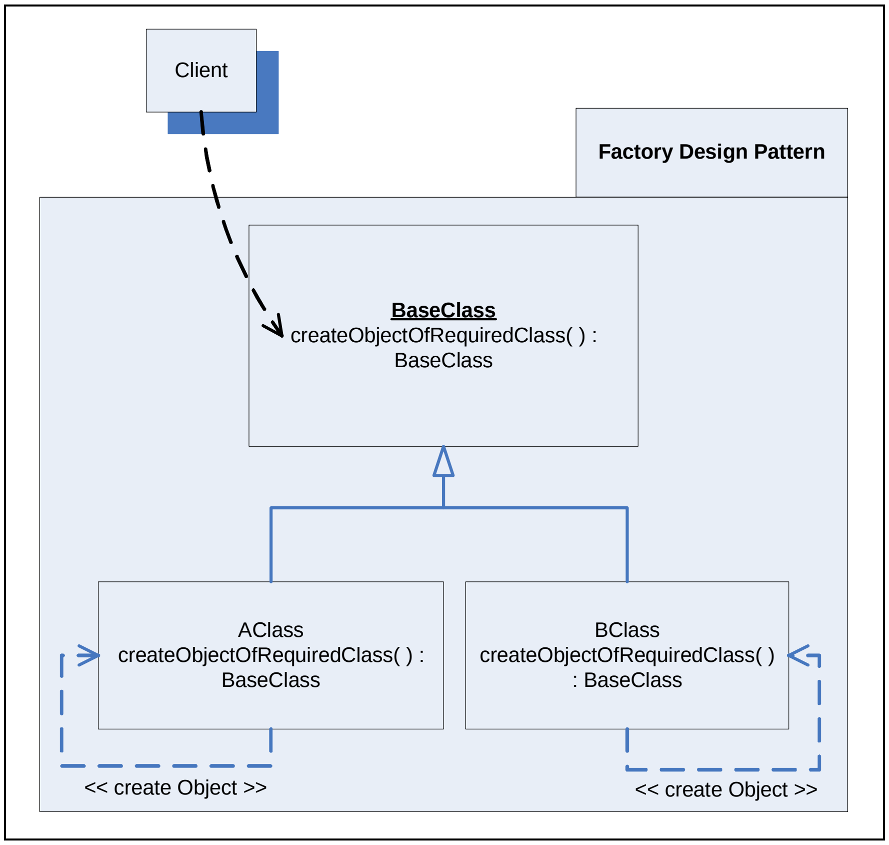

Client
Factory Design Pattern
BaseClass
createObjectOfRequiredClass( ) : BaseClass
AClass
createObjectOfRequiredClass( ) : BaseClass
BClass
createObjectOfRequiredClass( ) : BaseClass
<< create Object >>
<< create Object >>

Figure 18 Factory Design Pattern Class Diagram (From [20])

The interaction among these classes in the factory design pattern is illustrated in Figure 19. The client will contact the factory method—*createObjectOfRequiredClass*—when it needs to create an object. This interface will delegate it to the rightful concrete subclass at runtime, which has the know-how of instantiating the object.

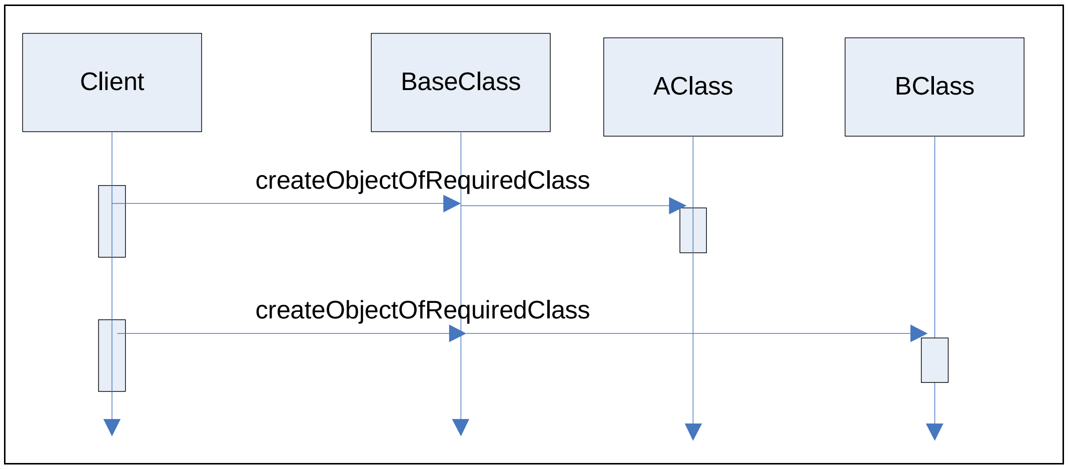


Figure 19 Factory-Design-Pattern Interaction Diagram

B. SINGLETON DESIGN PATTERN

1. Situation

A class in object-oriented programming defines the abstract characteristics that are common among class instances (objects) instantiated from the same class. These objects share the same kind of attributes or properties, and behavior. Each object owns its own unique set of attributes or properties while sharing some common class-level characteristics. There are, however, situations where there should be one, and only one, class instance of a specific class that should exist in the system, and all clients that need to contact an object of this class should be directed to the same object throughout the system. One approach would be to assign a managerial object the responsibility of instantiating the object of this class so that all clients will be able to access this one-and-only object. Unfortunately, this approach is unable to prevent accidental instantiation of an object, and in addition, has inevitably forced every client to be dependent on the managerial object. While insisting on the need for "only-one" objects, there is also the need for a clean dependency among clients in contacting this common object and avoiding accidental instantiations in the system. In this predicament, the design of the Singleton pattern evolved.

2. Intent

The intent of using a Singleton design pattern is to create a Singleton class that will ensure that there is one, and only one, object that will exist in the system, provide a global means of access to this common object, and prevent accidental instantiation. The Singleton pattern is applicable in situations where clients throughout the system need a common means of access [1]—a publicly available method—to this object. It is also applicable in situations where the Singleton class not only takes sole ownership of, and responsibility for, instantiating and deleting the common object, but ensures that only the Singleton has the ability to instantiate, denying any other possibility of instantiation.

3. Design Structure

The basic class design of the Singleton design pattern is illustrated in Figure 20. The *S* class is the Singleton class. The Singleton owns the one-and-only object that it instantiates by means of static persistency for the common object and provides a globally available method—*getSingleton*—for all clients to access, as illustrated by the dotted arrow pointing into the Singleton *S* class. The constructor of the Singleton class is not available to any client except the Singleton class itself; as a result, no accidental instantiation by other clients can take place. The compiler would have caught it.

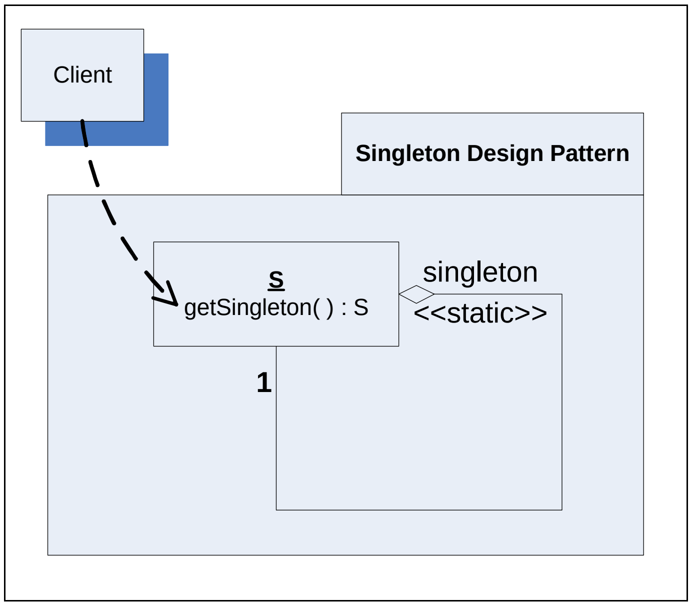


Figure 20 Singleton-Design-Pattern Class Diagram (From: [20])

IV. BEHAVIORAL DESIGN PATTERN

This chapter conducts detailed discussion of two design patterns that belong to the behavioral category. The approach is to focus on the situation that gives rise to each design pattern, its applicability and its design structure, to illustrate its characteristics. These two design patterns have also been suitably applied in the design work of this research.

A. OBSERVER DESIGN PATTERN

1. Situation

Most systems have requirements and design [20] that need data to be computed and the updated data to be presented in one form or another, whether on display or some other medium, or even simultaneously to be reflected and made available to several other means that the system supports. In a system where software modules are well organized, modules that compute and update data are known as the data source [20] or the subject [1] of interest. Modules that display the latest updated data are known as the observers [20] [1] of the data source [20] or subject [1]. Observers need to pay attention and observe the subject, because the latest update will need to be picked up almost instantly. There is a dependency [1] of the observer towards the subject. In this dependency, however, there should be no limitation on the number of observers that can observe a subject. Similarly there should not be any limitation on the number of subjects that an observer can observe. While there is a dependent relationship between a subject and its observer, they should not be tightly coupled so as not to reduce their reusability [1].

One straightforward approach to get the latest updated data is for each observer to constantly check and query the subject. This is the “poll” approach. But one can imagine the system inefficiency when the subject changes once in a long while, and the many observers making multiple checks find disappointingly unchanged information; the system becomes bogged down with unfruitful check and query transactions. An alternative approach is for the subject to provide notification when it has updated its data. This is the “push” approach. An observer establishes its dependency with the subject of interest—and as, and when, there is a change in data, the subject notifies all its observers. This second approach is the more elegant and is that which the observer-design pattern is based on.

2. Intent

The intent of the observer-design pattern is to define and establish a one-to-many [1] dependence between the subject and its observers, such that when one object—the subject—changes state, all its dependents—the observers—are notified and updated automatically in an efficient fashion. The dependency between an observer and its subject provides the loose coupling necessary to ensure that each retains reusability. This design pattern applies in situations when a change to one object is needed to trigger awareness of the change in other objects. The object triggering the change need not know which and how many objects are dependent on its change. This design pattern is particularly useful when there is a need to ensure loose coupling between objects that are dependent on each other.

3. Design Structure

The design of classes of the observer design pattern is illustrated in Figure 21. *Subject* and *Observer* are abstract classes. The dependency of *Observer* on *Subject* is reflected by the containment relationship that links the subject to its observer. *Client* accesses *Subject* to request that *Observer* to be notified of any changes made by the subject [20]. This is illustrated by the dotted line into the *Subject* class. *ConcreteObserver* is the concrete class that is interested in any notification of data updates by the subject. *ConcreteSubject* is the concrete class that houses the data source and assumes the role of data updater. In its updates, *ConcreteSubject* accesses the notification method that will inform every interested *Observer*.

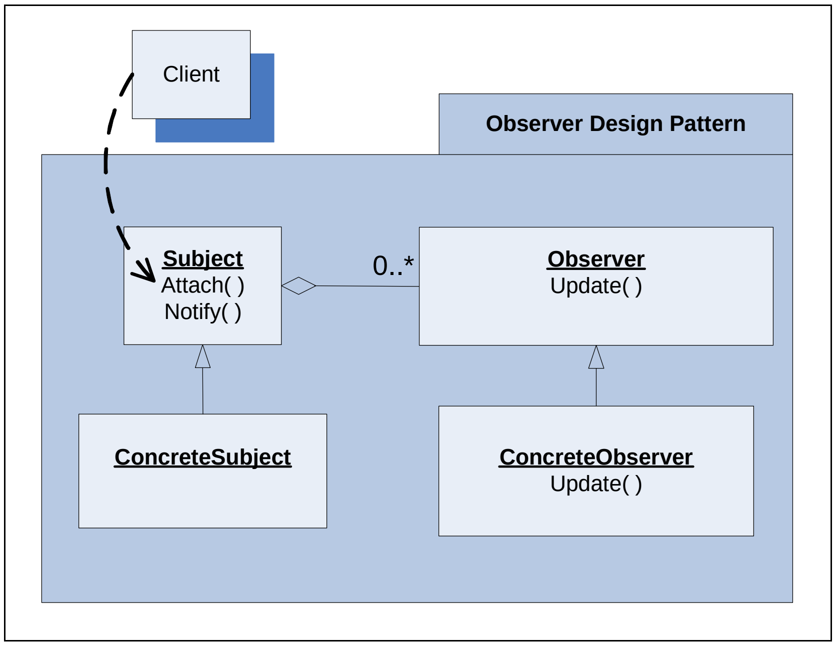


Figure 21 Observer Design Pattern Class Diagram (From: [1])

The interactions among the classes in the observer design pattern are illustrated in Figure 22. The *Client* will make the request to the *Subject* that there is an interested *Observer*—*ConcreteObserver*. When the *ConcreteSubject* effects an update on the data source, it notifies the *Observer*. It is the *Subject* that will update the *ConcreteObserver*. In fact, there could be as many *ConcreteObservers* that have established the interest in the *ConcreteSubject*. The *Subject* in this design pattern assumes the role of responding to as many interested *Observers*. The *ConcreteSubject* is unaware of and not bothered by who and how many interested *Observers* there are. This design decouples *ConcreteSubject* away from *ConcreteObserver*, creating a weak coupling for the dependency between *ConcreteObserver* on *ConcreteSubject*, so that the reusability of both *ConcreteObserver* and *ConcreteSubject* is not reduced.

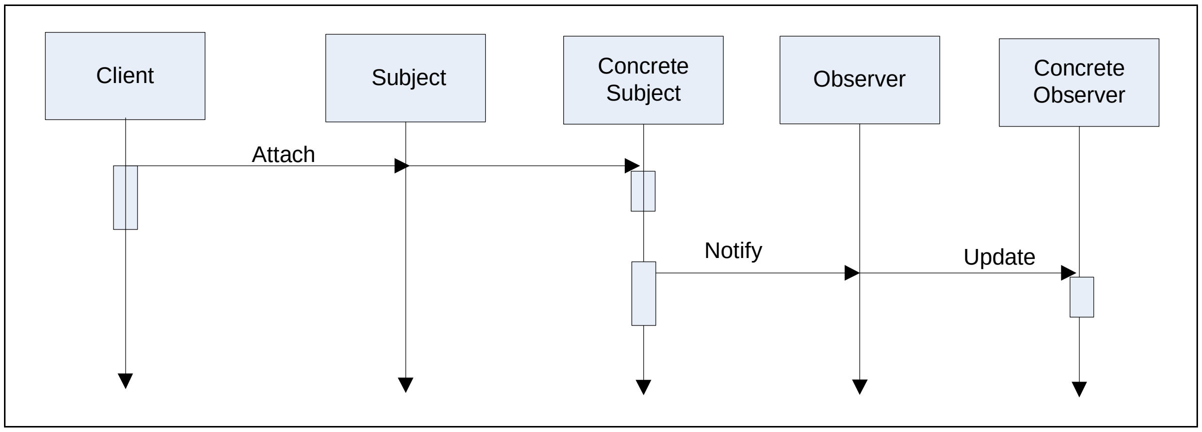


Figure 22 Observer Design Pattern Interaction Diagram

B. MEDIATOR DESIGN PATTERN

1. Situation

In small systems with only a few objects interacting, it is conventional that each object refers directly to other objects that it depends on. The straightforward referencing is clean and simple. Reflecting these references on a design document, each line of dependency will be readily visible. To reuse a small system module, these few classes will be used as is. When a system consists of several classes and many more objects interacting, the conventional approach when objects refer to other objects that they depend on directly is unfortunately not clean and simple. Direct referencing of objects reveals lots of interconnection between objects and the objects they depend on. Reflecting this straightforward referencing graphically on a design document would show cluttered cobwebs of dependencies. Such a system becomes monolithic [1]. It becomes difficult to change the behavior of the system when behavior is distributed among straightforward, but complex, interconnected classes and objects. The inter-referencing inhibits selected classes of behavior from being reused alone [20].

One approach to restoring simplicity would be to find some intermediary object that collects and consolidates the dependencies on other objects. This object will be dependent in turn only on its intermediary object. Different collections of object dependencies can be abstracted, such that each object will have a direct reference or dependency on the intermediary object, while the intermediary will be aware of the relevant dependencies among other objects and classes. This is the situation the mediator design pattern was created for. A mediator is the intermediary object that encapsulates collective behavior and is responsible for coordinating the interactions of a group of objects. Every object that needs to reference other objects will be referencing only the mediator. The mediator keeps interdependent objects within a group from referring to each other directly and explicitly, reducing the interconnectivity of lines of dependencies.

2. Intent

The intent of the mediator design pattern is to define an intermediary object—the mediator—that encapsulates and addresses how a set of objects will be interacting [1]. This design promotes loose coupling, as it discourages objects from referring to each other explicitly [1]. Abstracting and encapsulating the dependencies within the mediator, this design pattern promotes the flexibility to vary interaction among objects independently. In fact, as a third party, the mediator object aggregates referencing to other objects such that there is indirect dependency among these objects. This design pattern is applicable when an object needs to communicate with other objects in well-defined yet complex ways. It is useful to apply this design pattern on a seemingly unstructured set of interdependencies among objects of a system. With adequate abstraction and encapsulation of collective dependency behavior into the mediator, the behavior of the system becomes easily customizable as objects and classes become more independently reusable.

3. Design Structure

The design of classes of the mediator design pattern is illustrated in Figure 23. The *Colleague* abstract class and *Mediator* abstract class define the single simple and clean dependency of a *Colleague* object on a *Mediator* object. *ConcreteColleague_1* through *ConcreteColleague_n* are concrete classes whose objects have interdependencies. The *ConcreteMediator* is a concrete class that aggregates the references to all the interdependent concrete classes. An object of the *ConcreteMediator* class will be aware of the dependencies for its respective *Colleague* object. In this way, this design provides the means to flexibly vary the interactivity among objects. While the

ConcreteMediator class in this design facilitates the flexibility of defining dependencies among objects, *ConcreteColleague_1* through *ConcreteColleague_n* class, each of which defines unique behaviors, they are now individually more customizable and independently reusable.

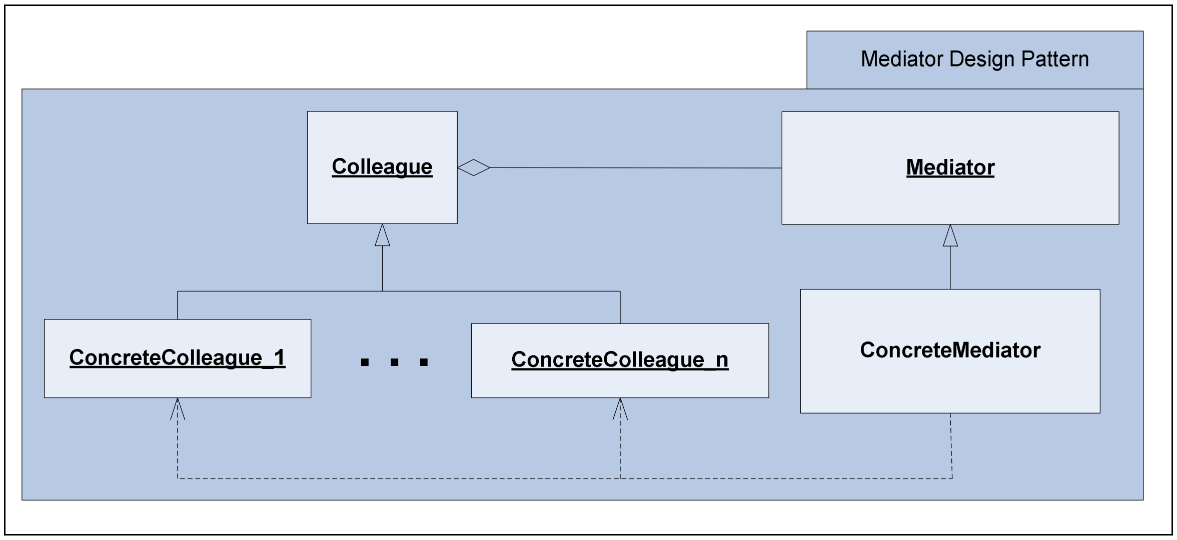


Figure 23 Mediator Design Pattern Class Diagram (After: [20])

The interaction among classes of the mediator design pattern is illustrated in Figure 24. With the containment relationship that establishes the dependency of the *Colleague* abstract class on the *Mediator* abstract class, an object of *ConcreteColleague_1* would reference its *ConcreteMediator* object, which would in turn reference other objects that the object of *ConcreteObject_1* has dependencies on. *Concrete_Colleague_1* till *ConcreteColleague_n* are concrete classes whose respective objects also have the access to the mediating role of their respective *ConcreteMediator* object.

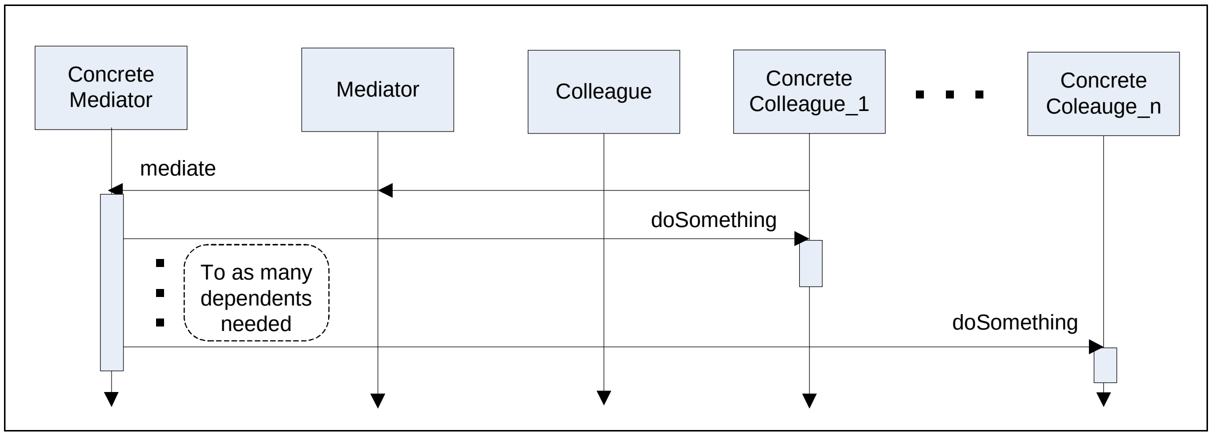


Figure 24 Mediator Design Pattern Interaction Diagram (After: [20])

THIS PAGE INTENTIONALLY LEFT BLANK

V. JAVA LISTENER MECHANISM

This chapter looks at two kinds of listener mechanisms in Java. The discussion focuses on understanding the concept and design of each Java listener mechanism and identifies what design patterns have actually been incorporated. This is helpful in the understanding of what and how design patterns have been used while Simkit's modeling paradigm advocates its concept of listener mechanism.

A. ACTION-LISTENER MECHANISM

1. Concept

The *action listener* is a Java interface for receiving an action event [21], that is, a specific kind of event that reflects the occurrence [21] of a component action that the Java framework supports. In Java framework, an event is represented by an object that gives information about the event and identifies the event source [32]. Event sources are often components, models, or any other kind of object that can be an event source [32]. An object capable of generating events is the event source, while an object interested in the events of an event source is an event listener. An event source can be associated with multiple event listeners. Similarly, an event listener can be associated with multiple event sources. This many-to-many relationship between event source and event listener is the event model concept of Java framework [32]. This model is inherent in each of the different kinds of events supported in Java framework, including the action event—event source—and its corresponding action listener—event listener.

A *component*, specifically the *Component* class in Java framework, is defined as an object that has a graphical representation that can be displayed on the user interface (UI) and can interact with the user, capturing user-specific actions [21]. Each action, e.g., mouse clicks and keyboard entries, that the UI component accomplishes with the user is a component action that is encapsulated in an action-event object. The component is the event source of action events. Any object whose class implements the action interface will be able to register itself as an action listener with the component that generates action events. Whenever a user-specific action occurs, the action-listener object is notified of the user action in the form of an action event. In this way, the action listener is given the chance to encapsulate application-specific functionality and behavior in response to user-specific actions, through the UI component. In the Java framework, there are several UI components that can capture user interactions into an action event. They are the *Button* and *List* components.

A class—specifically, an application class—whose objects need to register as action listeners will have to define the *ActionPerformed* method, as specified by the action-listener interface. The information in a user-specific action that is captured as an action event will be available to the application through this method as a parameter, allowing application behavior to respond to user actions as required. This is the mechanism that Java framework uses to facilitate the application's defining behaviors in response to user actions.

2. Design Structure

The class design in the Java framework that supports the action-listener interface is illustrated in Figure 25. The *ActionListener* interface belongs to the *Java.aw*t package. This interface is a subclass of the *EventListener* interface that belongs to the *Java.util* package. The *Button* and *List* are UI Component classes that subclass from *Component* class. These classes are part of the pre-defined set of UI *Component* classes that Java framework has created and grouped under the *Java.awt* package. The *ActionEvent* class that is a subclass of the *AWTEvent* class is defined and grouped in *Java.awt.event* package. There is a dependency by the *ActionListener* interface on this *ActionEvent* class. The *Application* uses the *Java.awt* package and implements the *ActionListener* interface. Each event source—the *Button* and *List* objects respectively—can be associated with many event listeners by *ActionListener*. The event listener—the *Application* object—can be associated with many event sources—whether *Button* objects or *List* objects—independently.

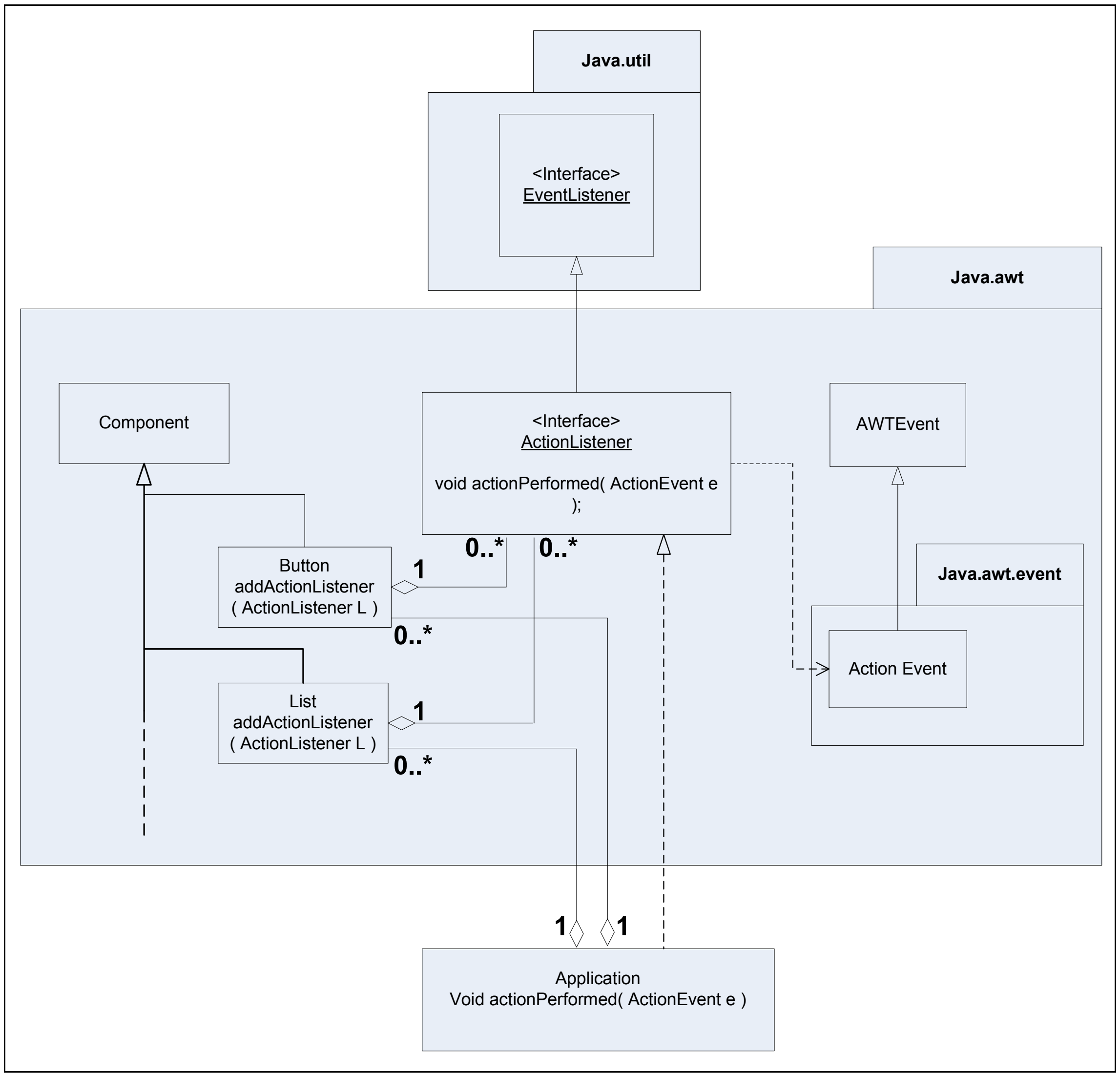


Figure 25 Java Action-Listener Class Diagram

3. Incorporated Design Patterns

The mechanisms inside the Java framework make abundant use of interfaces. The Java action-listener mechanism discussed earlier has illustrated how its interface is used in the design of its mechanism. Although the design of design patterns involved only OO classes, by analyzing the behavior of the Java action-listener mechanism and design patterns, the presence of design-pattern behavior in action-listener mechanisms can be identified. The action-listener mechanism of the Java framework has in fact incorporated two design patterns: the observer and the mediator.

The analysis of the Java action-listener mechanism where its behavior incorporated the observer design pattern is illustrated in Figure 26. In the observer design pattern, the concrete observer will establish its link with the concrete subject containing the data of interest through an attachment setup process. In Java's action-listener mechanism, the application will also need to establish its link with the UI component through the action-listener registration-setup process.

During runtime, the concrete observer is notified (through its update method) by the concrete subject, whenever this subject of interest has an updated data. This behavior has its correspondence in the Java action listener where the application's action listener is notified through its *ActionPerformed* method by UI component *Button* when a user-specific action occurred. Both the observer design pattern and the action-listener mechanism push out the change to the concrete observers and application, respectively.

As the concrete subject pushes out notification to its concrete observers, it is unaware of the number of concrete observers and independent of each specific concrete observer. In similar fashion, the UI component *Button* is aware neither of the number of action listeners it needs to notify nor of the specific action listener it is notifying.

This relationship has allowed UI components to be developed independently from the application, and there is optimal reusability of UI components across different applications. The independence of the UI component and the application reflects the loose-coupling characteristics that the observer DP advocates between the concrete subject and its concrete observers, in order to facilitate independence and ensure high reusability.

Figure 26 illustrates the key classes—*Button* and *Application*—and interface—*ActionListener*—of the Java action-listener mechanism and the corresponding classes—*Subject*, *ConcreteSubject*, *Observer* and *ConcreteObserver*—from the observer design pattern that the mechanism has incorporated. Figure 26 also illustrates the relationships among the classes that reflect the behavior of the observer design pattern that has been identified.

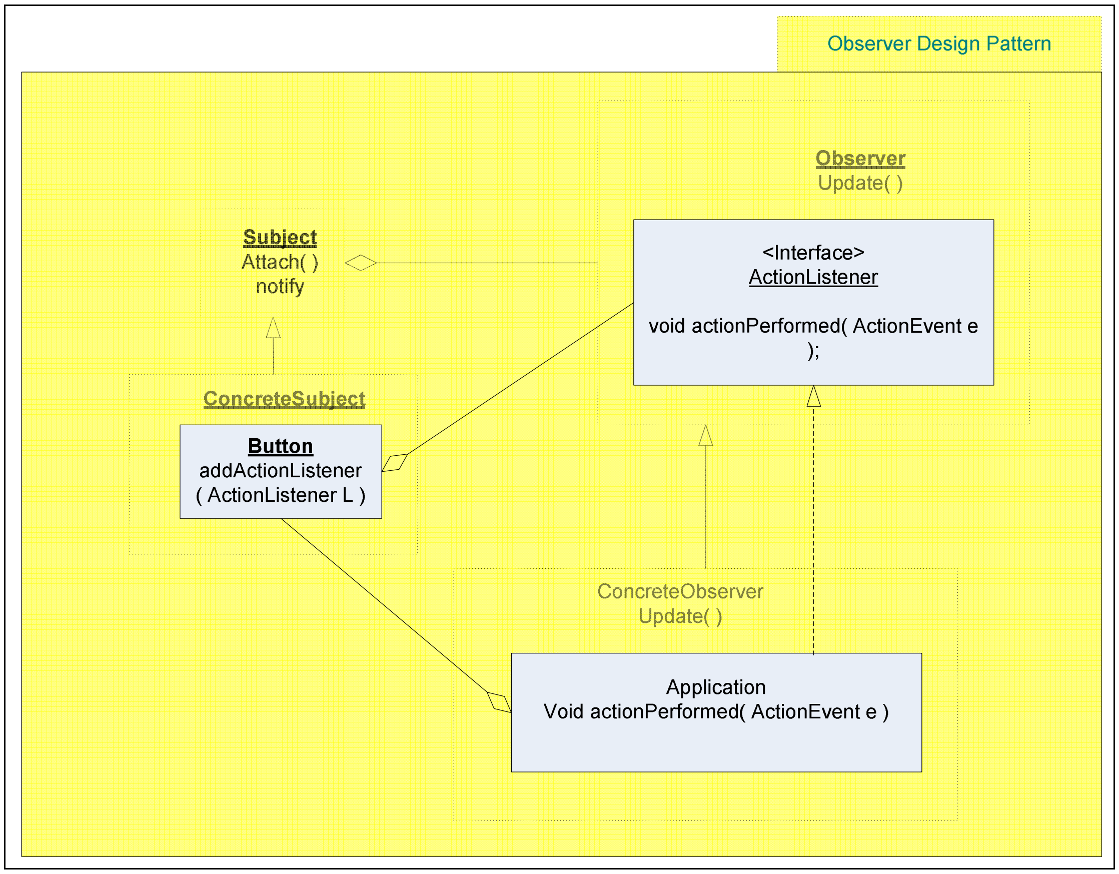


Figure 26 Observer Design Pattern in Java Action-Listener Mechanism

The analysis of the Java action-listener mechanism where its behavior incorporates [20] the mediator DP is illustrated in Figure 27. In the mediator pattern, once it is set up, each concrete colleague—*ConcreteColleague1* and *ConcreteColleague2*—contacts the mediator individually and independently. In the Java action listener, the UI components—*Button* and *List*—contacts the *ActionListener* individually and independently once the setup is completed.

The mediator design pattern advocates that an intermediate object will collect and consolidate the dependencies among objects that are interdependent on each other. This is the design that reduces the interconnectivity among these interdependent objects. In this way, only *ConcreteMediate* is aware of interactions among these interdependent objects. *ConcreteMediate* also has the flexibility to vary the interactivity among objects. This behavior can be identified in the design of the Java action-listener mechanism, which advocates that the application define the behavior of how the UI components will be affected as a response to different user-specific actions from each UI component. Each application has the flexibility to vary the interactivity among interdependent UI components. The role of the application collects and consolidates the interactivity among dependent UI components.

Although each UI component may be dependent on other UI components, the *Application* class mediates their dependencies. In this way, *Components* is loosely coupled and independently reusable. This loose coupling reflects the characteristic that the mediator design pattern advocates, which discourages direct referencing among interdependent objects.

Figure 27 illustrates the key classes—*Component*, *List*, *Button* and *Application*—and the interface—*ActionListener*—of the action listener and the corresponding classes—*Colleague,* *ConcreteColleague1*, *ConcreteColleague2*, *ConcreteMediator* and *Mediator*—of the mediator design pattern the mechanism has incorporated. Figure 27 also illustrates the relationships among the classes that reflect the behavior of the mediator DP.

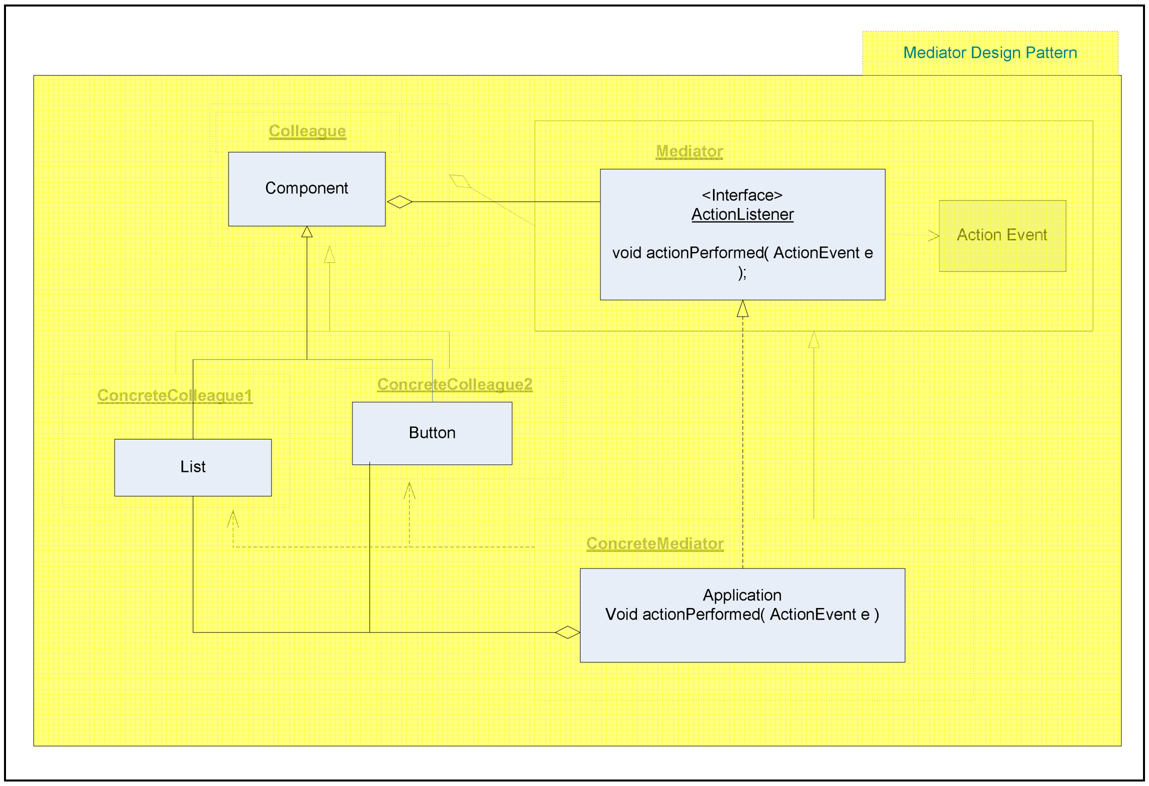


Figure 27 Mediator Design Pattern in Java's Action-Listener Mechanism

B. PROPERTY-CHANGE-LISTENER MECHANISM

1. Concept

The property-change listener is a Java interface that receives the property-change event when the Java bean has been updated. In Java, JavaBeans is defined as a portable, platform-independent, component model written in the Java programming language [22]. JavaBeans architecture advocates that JavaBeans ("bean" in short) has to be a reusable, portable and platform-independent component that can be used in applets, java applications, and in building composite component. The JavaBeans specification indicates that the dynamic nature of bean will support the use of property sheet or a bean customizer, such that bean's property can be customized and modified in design mode. Most Java components are built to meet the JavaBeans specification. All UI components in Java framework are beans. In Java, there exist non-UI components that are also beans.

They can be discovered, customized, and modified through the property sheet or beans customizer. When a change in the property of a bean occurs, a property-change event will be created to capture information about the change.

In JavaBeans architecture, a property change is an event that is created when there is a change in the “bounded” or “constrained” property of a Java bean [21]. This event contains specific information about the bean: its name, the new value that was updated and that resulted in the creation of this event, and the previous value [21]. JavaBeans architecture adopts the event model of the Java framework. The bean is the event source. It is capable of generating property-change events. The property-change listener is interested in receiving property-change events. The many-to-many relationship of the event model is inherited in the JavaBeans architecture between the bean and its property-change listener.

A class—namely, the *Application* class—where the objects are interested in receiving the property-change event notification from a Java bean will need to define the *PropertyChanged* method, as specified in the property-change-listener interface. The information about the update on the property of the bean will be available to the application in the parameter, allowing the application to respond to bean changes. This is the mechanism that allows *Application* to define its behavior in using beans.

2. Design Structure

The class design in the JavaBeans architecture that supports property-change listeners is illustrated in Figure 28. The *PropertyChangeListener* interface belongs to the *Java.beans* package. It is a subclass of the *EventListener* interface from the *Java.util* package. It has a dependency on the *PropertyChangeEvent* class that resides in the same *Java.beans* package. The application uses the *Java.beans* package and implements the *PropertyChangeListener* interface. In the Java framework, all UI components are beans. There are also non-UI components defined in the *Java.awt* package. These components are the event source. The application is the event listener.

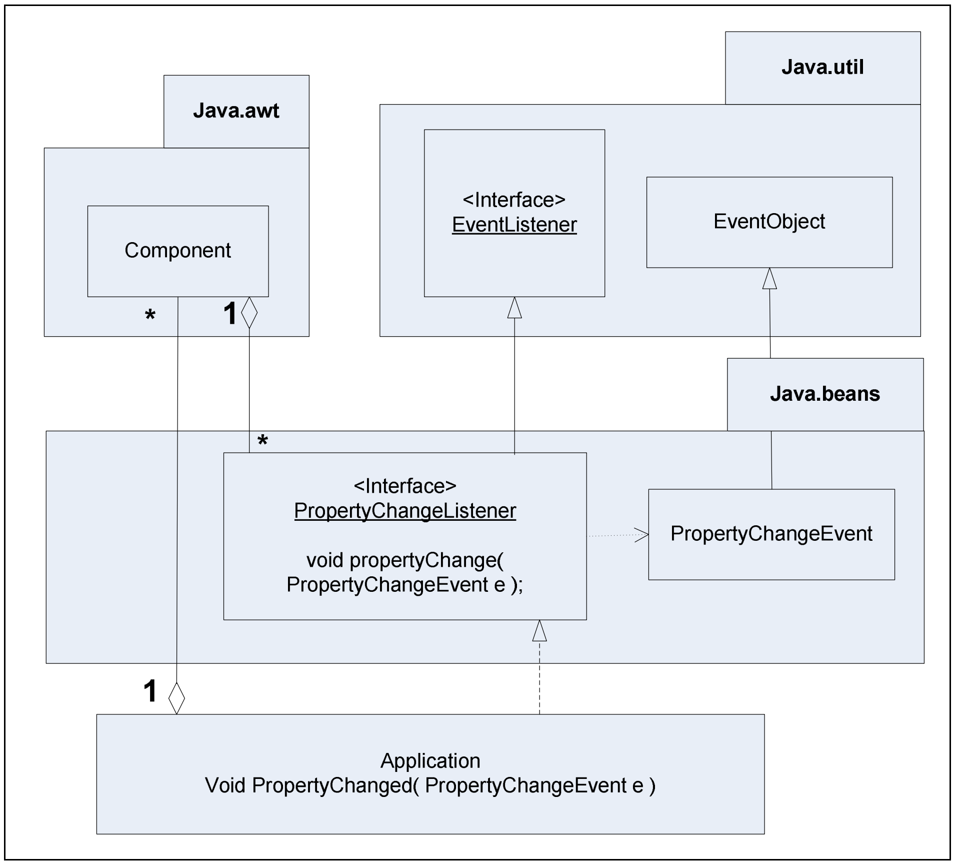


Figure 28 Java's Property-Change-Listener Class

3. Incorporated Design Pattern

The behavior of the property-change listener shares familiar behavior with design patterns. Analysis of the design and behavior of the property-change listener has identified that it has incorporated the observer DP.

Figure 29 illustrates analysis of where the observer design pattern has been incorporated into the property-change-listener mechanism. In the observer DP, the concrete observer establishes its link with the concrete subject that contains the data of interest through an attachment-setup process. In the property-change mechanism of the JavaBeans architecture, the application that implements the *PropertyChangeListener* interface will need to register with the interested *Component* directly. This sets up the object of the *Application* to receive a property-change event when that *Component* has an update [21].

During runtime, the concrete observer is notified of an updated data by the concrete subject, through its *update* method. In similar fashion, when *Component* updates its property, it notifies *Application* through its *PropertyChange* method. Both the observer design pattern and the property-change listener mechanism push out the change to the concrete observers and application.

In pushing out notification of change to concrete observers, the concrete subject is unaware of the number of concrete observers and independent of any specific concrete observer. Similarly, each *Component* of the JavaBeans architecture is not aware of the number of property-change listeners it needs to notify, nor of specific property-change listeners.

The mechanism adopted by the JavaBean architecture allows the bean (*Components*) to be developed independently from the application. In addition to that, the JavaBeans specification also addresses reusability, portability, and platform-independence across applets, applications, and composite-component building. This independence between the bean and property-change listeners is possible because of loose coupling in the dependency relationship, which is the characteristic the observer DP advocates for high reusability of components.

Figure 29 illustrates the key classes—*Component* and *Application*—and interface—*PropertyChangeListener*—of the property-change-listener mechanism and the corresponding classes—*ConcreteSubject*, *ConcreteObserver* and *Observer*—from the observer DP that the mechanism has incorporated.

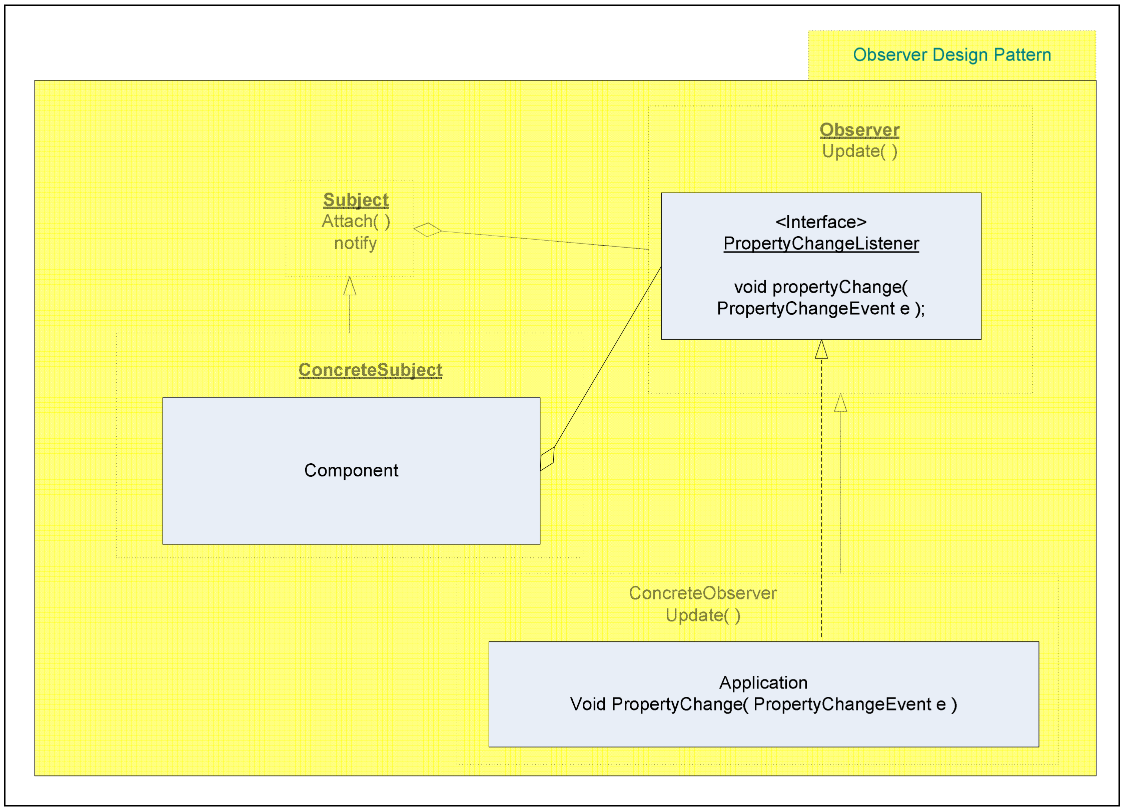


Figure 29 Observer Design Pattern in Property-Change-Listener Mechanism

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CURRENT DESIGN PATTERNS IN SIMKIT

This chapter focuses on design in the random utility package, the intra-component and inter-component mechanism of Simkit. The design of key features of each module will be illustrated in detail, followed by an analysis of the design implemented.

A. RANDOM NUMBER

The use of random variates is common in many applications. In collecting statistical results for analysis from multiple simulated runs, the use of reliable random variate generators plays an important role. Simkit has a module that makes random variates easily available to applications from the framework. This module allows new random variates to be implemented to determine the required reliability of the source of number generation. This section will look at the design that makes this possible and conducts an analysis of the design as implemented in Simkit.

1. Design

A common approach most simulation applications adopt is the use of random variates in their computational models. There are different characteristics of random-variate generators. Each random-variate generator has parameters that can create variations in a random-number set. An application can build its own random-variate generator utility or use a random-variate-generator utility so it can focus on the logic of its business model. Applications are more interested in using random variates than how these numbers are created.

Simkit has a suite of useful utilities to support the development of a wide variety of simulation applications, including a package for generating different types of random variates. This package is used extensively in Simkit-based applications spanning a wide range of application domains and is particularly useful when a repetitive simulation run is needed for statistical analysis.

The way classes are designed in the random package made it one of the independent Simkit utilities that support rapid development. Not only is it independent of other software packages, it can even be used by non-Simkit Java applications that are only interested in harnessing the random-number generation functionality. This supports the ease with which this whole package can be replaced. The package contains a rich set of different types of random number generators and an extensive variation of random variates. The design of classes within this package supports an easy approach in which new random-number generators and new random variates for future applications can be easily created, customized and expanded.

In the random package, random-number generators can be created through *RandomNumberFactory*, while random variates can be created through *RandomVariateFactory*. *RandomVariateBase* provides an abstraction that facilitates different variations of random number set to be defined. This is a simple approach where different variations of random number sets can be created. *RandomVariateFactory* in fact uses random numbers created from *RandomNumberFactory*. *RandomVariateFactory* has a dependency on *RandomNumberFactory*. This is illustrated in Figure 30. It also shows that the *RandomVariateFactory* has a dependency on the *RandomVariate* interface. This interface holds the key signature to which each variation of random number variate has to comply. Concrete random-number generators—concrete classes—can be created by implementing the *RandomVariate* interface directly. Another approach would be to subclass the *RandomVariateBase* abstract class. The relationship of the key interfaces and top-level abstract class in the random package is illustrated in Figure 30. For a detailed list of all the concrete classes and random-number variates that are available, see Appendix A.

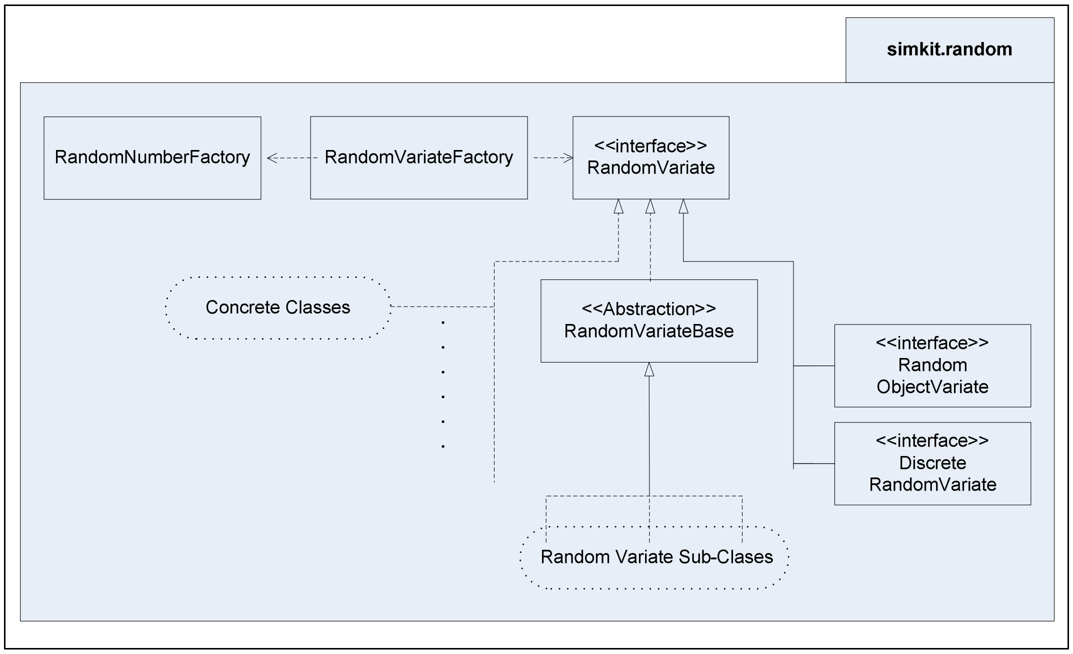


Figure 30 Simkit Random-Package Top-Level Class Diagram

The interaction that the *Client* application would conduct in using this random package to obtain a random variate is illustrated in Figure 31. The *Client* will make a single contact on the *RandomVariateFactory* to request for a *RandomVariate*. A new concrete subclass of *RandomVariateBase* (a concrete random variate) will be created and made available to the client. The client is thus equipped with a *RandomVariate* that will generate random numbers directly. The various steps that the *RandomVariateFactory* undertake, as shown in Figure 31 are encapsulated away from the client.

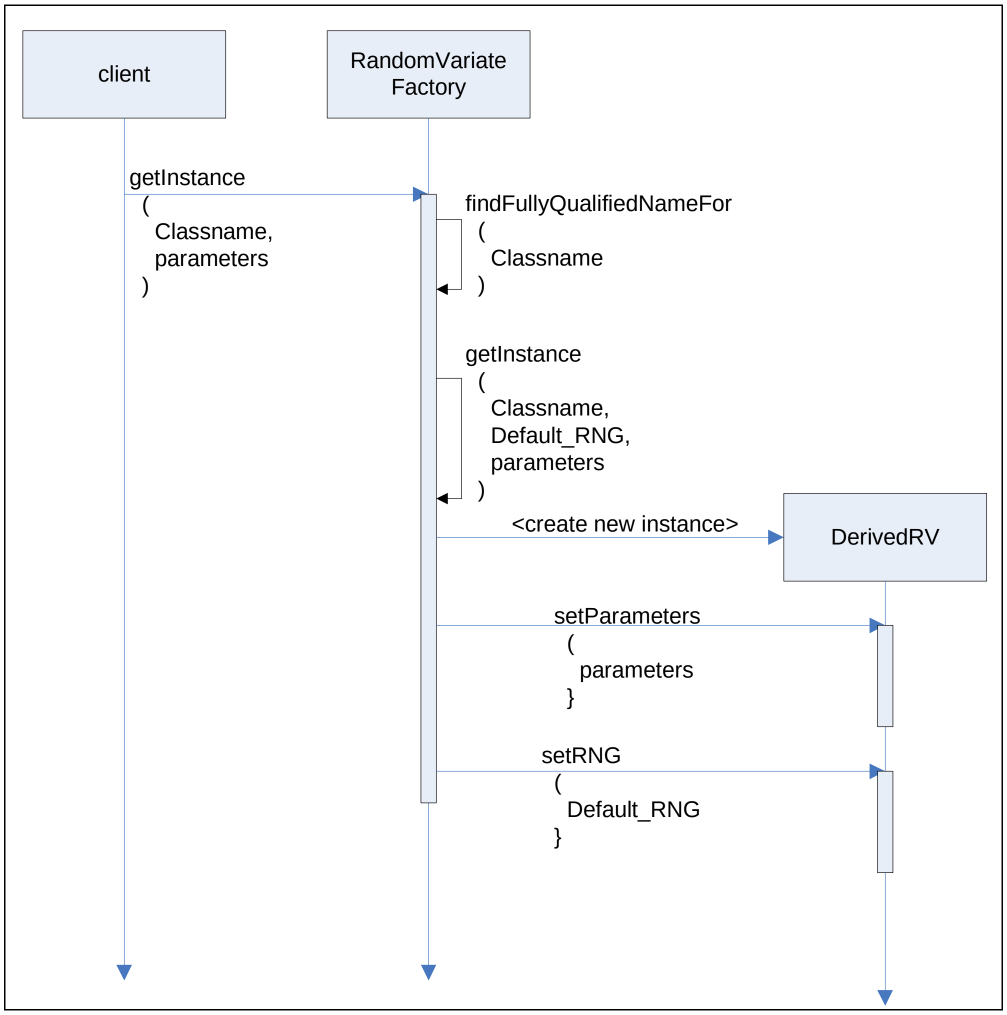

client
RandomVariate Factory
getInstance (Classname, parameters)
findFullyQualifiedNameFor (Classname)
getInstance (Classname, Default_RNG, parameters)
<create new instance>
DerivedRV
setParameters (parameters }
setRNG (Default_RNG }

Figure 31 Creating Random Variates using Simkit's Random-Package Sequence Diagram

2. Analysis

The random package is an easy approach where different characteristics of random-number generators can be created and different variations of the same random-number generator can be created easily. The development of new random-number generators and random variates can be carried out independently from the business logic of the application model. The flexibility and ease of quickly creating random-number generators and random variates allow this utility package to grow over time as it is used in a variety of applications.

This package also makes requests for random numbers and variates easy, allowing client applications to focus on their business logic.

Within a single process, there is a need for a single managerial object to administer a common means whereby random variates will be created and to manage this set of random variates. This single point of contact that manages all the random variates falls on the *RandomVariateFactory*. However, the first client that requests a random variate potentially incurs a high cost in terms of a long wait while the *RandomVariateFactory* discovers, loads and creates the concrete variate.

Although the *RandomVariateFactory* provides a single contact—the *getInstance* method—with which *Clients* can request for random variate, the name of the method is a misnomer. Figure 31 shows that the *Client* invokes the *getInstance* method of the *RandomVariateFactory* to request an object from *RandomVariate*. Unfortunately, the name of this method is inclined to suggest that an object of the *RandomVariateFactory* is being requested.

B. INTRA-COMPONENT EVENT SCHEDULING

A component in Simkit is independent because it has a set of self-contained event logic within the component itself. This is made possible through the mechanisms that facilitate intra-component event scheduling. This section of the chapter will look at the design of the mechanism that make this possible and conduct an analysis of this design that has been implemented in Simkit.

1. Design

In the discrete-event paradigm, the system is perceived as modeling system-state trajectories at the discrete occurrence of events. A running discrete-event simulation system is the continuous scheduling and execution of discrete events that propel its simulation execution over time. The modeling of a discrete-event system will need to define its own system-state variables and its set of events where these system-state trajectories will occur in the application.

Each application is different in the business logic of when its events will occur and the computational logic of system-state trajectories in each event. How an event should be scheduled and how the scheduled event should be triggered should be unanimous across all discrete-event systems. These are best addressed through a discrete-event simulation engine framework that will define the modeling paradigm and the underlying mechanism in scheduling and triggering events. System modeling by the applications will be able to leverage such a framework as they focus solely on their business logic, according to the requirements of their problem space.

Simkit is a discrete-simulation-engine framework that supports rapid development of a discrete-event-simulation application. At heart, the simulation engine defines mechanisms to support the scheduling of events and the triggering of scheduled events. An event—*SimEvent*—is an object that contains essential information about when this event is scheduled to occur, the data values that correspond to the arguments of this event, and the name of the event itself. Application modeling will schedule an event while Simkit ensures the creation of events and an ordered triggering of events scheduled. In triggering a scheduled event, the appropriate entry point of an entity will be triggered. An *entity* is an object whose class houses all event-execution logic. The event execution logic defines the behavior of the entity and is distributed among the methods of the entity's class. Each method—with a predefined prefix—where the business and computation logic resides is the entry point of a corresponding event. An entity in Simkit is an independent component in which all the required events and event-execution logic reside—intra-component—and all the methods of entity's class. This is the modeling paradigm of the Simkit framework.

Figure 32 illustrates the classes of the *simkit* package that enable the modeling paradigm of Simkit framework. An entity's class—*Application Derived Sim Entity*—will be a subclass of the *SimEntityBase* abstract class. This abstract class provides the mechanism to resolve the entry point that corresponds to each triggered event. An event will be scheduled by the entity of the *Application Derived Sim Entity* class. The mechanism of creating and scheduling a *SimEvent* resides in the *BasicSimEntity* abstract class. The *SimEntity* interface binds the entity to the event-triggering mechanism. The *Schedule* manages the state of the simulation run. The *EventList* is the contact point to schedule an event and also ensures ordered triggering of all scheduled events.

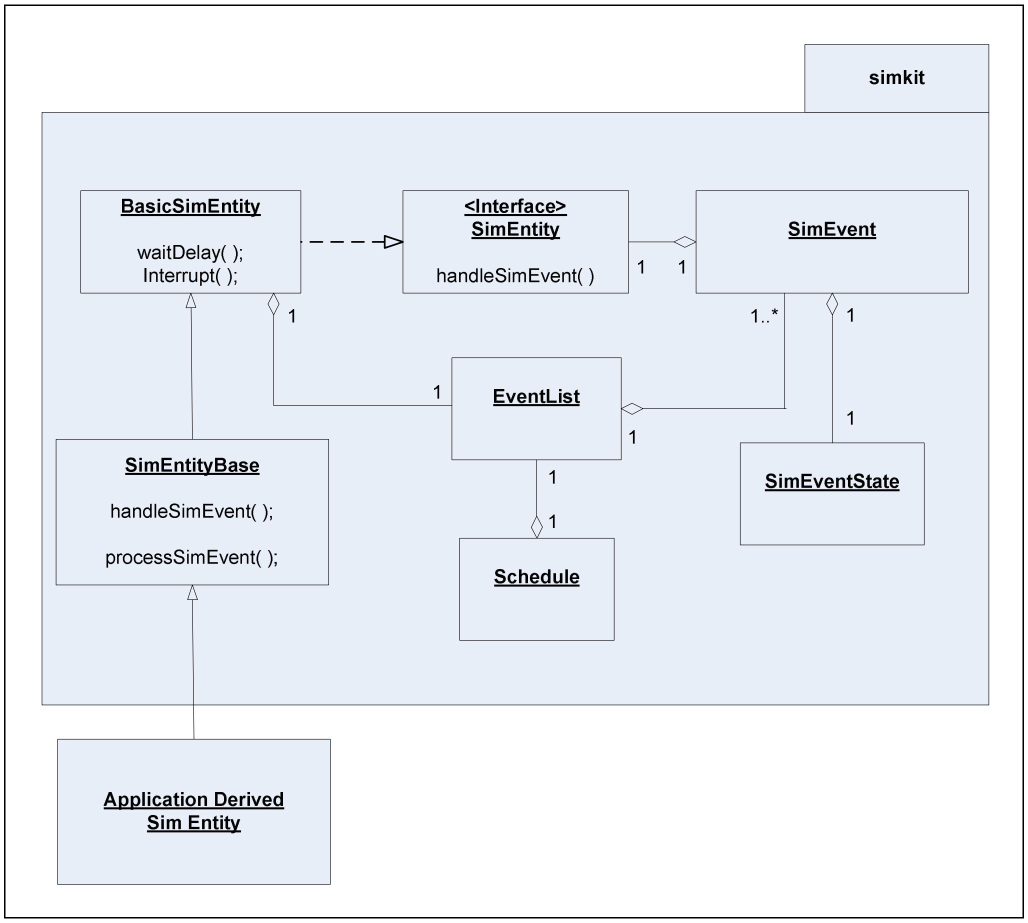


Figure 32 Simkit Event Scheduling Class Diagram

Figure 33 illustrates the interaction between the entity of the *Application Derived Sim Entity* class and the classes from *simkit* package and the mechanism that supports event scheduling. The business logic of the *Application Derived Sim Entity* will access the *waitDelay* method of the *BasicSimEntity* abstract class to schedule an event. A *SimEvent* will be created and added into the list of scheduled events in *EventList*. The corresponding cancellation of scheduled event is in Appendix B.

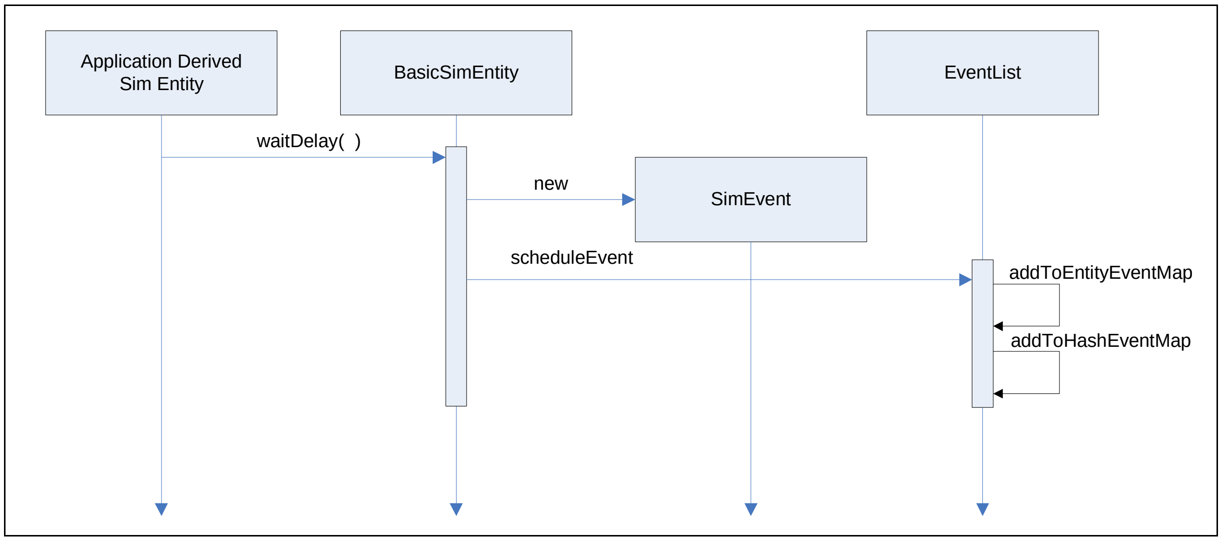


Figure 33 Event Scheduling Sequence Diagram

Figure 34 illustrates the mechanism in Simkit where an event is triggered. The first event will be removed—*popFirstEvent*—from the ordered event list. The *SimEntity* will be retrieved from the *SimEvent*. The interface acted as the abstraction where the *SimEntityBase* will be contacted to resolve the appropriate entry point on the *Application Derived Sim Entity* that corresponds to the event being triggered.

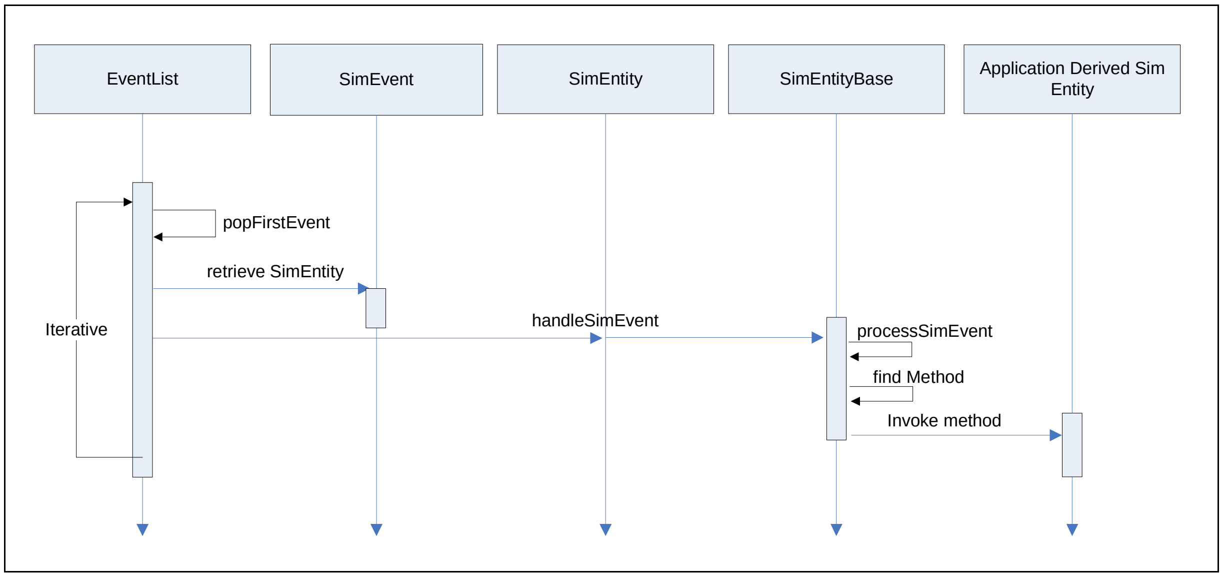


Figure 34 Event Triggering-Sequence Diagram

2. Analysis

A study of the class diagram in Figure 32 showed that the *SimEntity* is an interface that is well deployed in this design. This interface binds the event-triggering mechanism of the simulation engine to the entity, yet it elegantly decouples the simulation engine away from knowing the specific entity's class.

There is a *Schedule* class as shown in Figure 32. The name, "Schedule," indicates that this class has the role of scheduling and triggering every scheduled event. However, the class's role is only to take care of the state of the simulation system. The role of scheduling and triggering of events falls on *EventList*. In addition to these two roles, *EventList* stores and orders all scheduled events. It does seem that this class has been loaded with multiple roles and responsibilities. Future expansion on any functionality of this class will require painstaking effort.

In the modeling paradigm, using Simkit event execution has three distinct portions: retrieval, triggering, and execution. The entity (*Application Derived Sim Entity*) assumes the role of event execution as the computational logic of system state trajectories resides in the methods of *Application Derived Sim Entity* class. Simkit assumes the other two roles, of retrieving and triggering an event. Figure 34 showed that the entity (*SimEntityBase*) itself is retrieved from *SimEvent*. *SimEntityBase* then proceeds to resolve the entry point of the entity. It becomes apparent that *SimEvent* is a passive placeholder of the occurrence of an event, while *SimEntityBase* has an active role in the event triggering mechanism.

C. INTER-COMPONENT EVENT SCHEDULING

Independent components in Simkit can be combined to build larger, more complex system. This is made possible by the mechanism that facilitates chained interactivity of event scheduling across the components assembled. This section will look at the design that makes this possible and analyzes its implementation in Simkit.

1. Design

The modeling paradigm that Simkit advocates has led to the development of simulation applications that build entities that are independent components. Each entity has its set of event-graph logic. The simulation application has, in fact, comfortably built a set of independent components that each can be easily plugged into other applications when the component meets requirements. The autonomy of components would have elevated the modeling paradigm further if there were some means to link these components to effect some chained interactivity.

The LEGO [11] framework, where each independent component can be linked rapidly to build a larger complex system, has been incorporated in Simkit framework. Two independent components can be linked together where the event-triggering mechanism on the source component can be propagated to trigger events on the listening component. This is a simple high-level understanding on how the component-listener mechanism works. LEGO [11] framework allowed as many listening components as required to link to a source component. A component assumes duality as a listener on some components and as the source component to others. The semantics have been described in Chapter II, C. Simkit. The design in Simkit for LEGO [11] framework extends the event-triggering mechanism across components—inter-componently—through the listener mechanism. This is the approach the modeling paradigm Simkit adopts towards component-based simulation modeling.

Figure 35 illustrates the additional class and interfaces in *simkit* package that implements the listener mechanism for the LEGO [11] framework. The *SimEventSource* interface specifies the method (*notifyListeners*) that the *BasicSimEventSource* will implement to inform all its listener components when a scheduled event has occurred. The *SimEventListener* interface specifies the method (*processSimEvent)* that the *SimEntityBase* will implement to trigger events as a listening component.

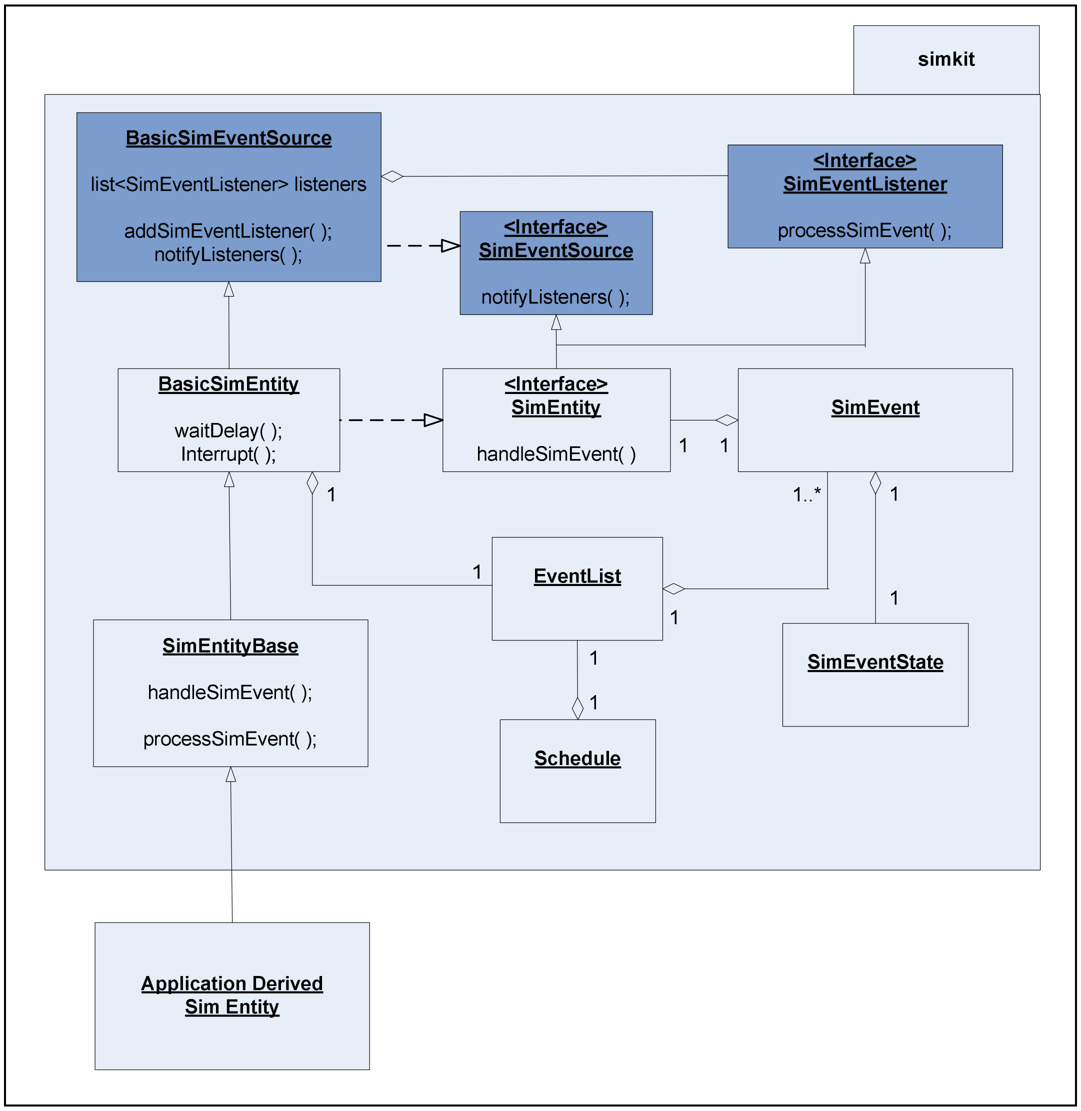


Figure 35 Simkit Component-Listener-Class Diagram

Figure 36 illustrates the interaction between the entity of the *Application Derived Sim Entity* class and the *BasicSimEventSource* class from *simkit* package. The entity—source component—adds a listening component that listens to its triggered event. The corresponding removal of a listener component is in Appendix C.

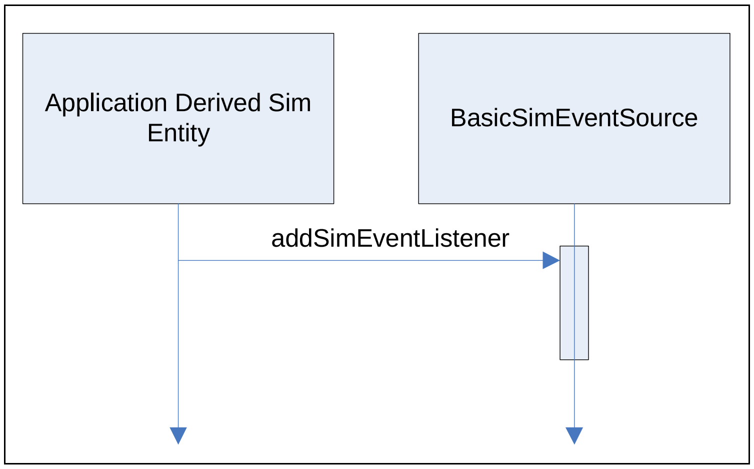


Figure 36 Simkit Adding Component Listener Sequence Diagram

Figure 37 illustrates the mechanism in Simkit where a scheduled event occurred; it is propagated to its listener components for event triggering. The event is removed as usual. The *SimEntity* is retrieved from *SimEvent*. The same interface now acts as the abstraction where the *BasicSimEventSource* will be contacted. The appropriate entry point on the *Application Derived Sim Entity*—listener component—that corresponds to the occurring event will be resolved for event execution.

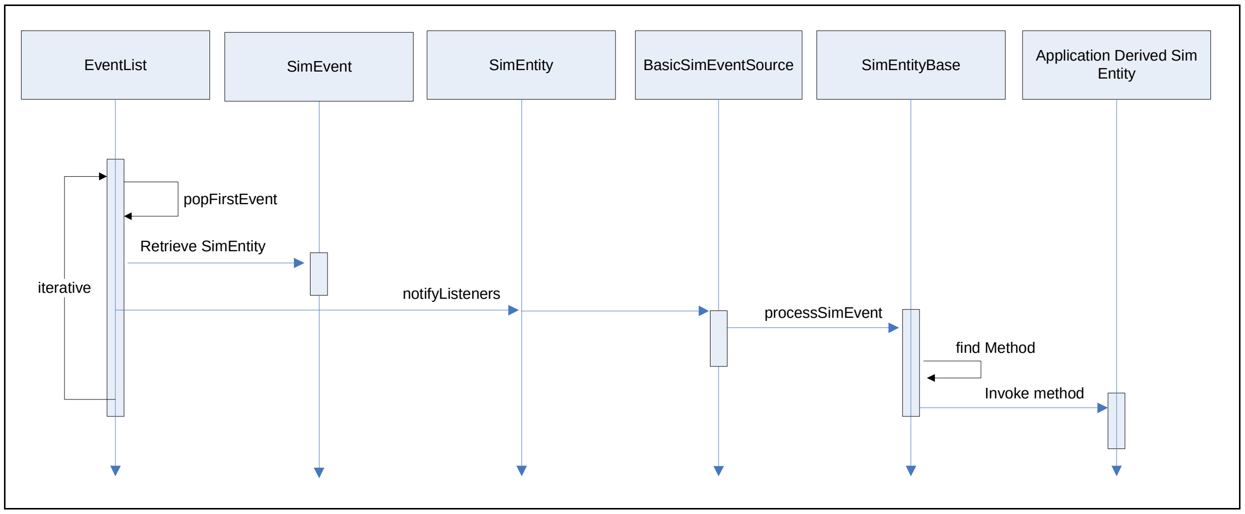


Figure 37 Simkit Triggering-Component-Listener-Sequence Diagram

2. Analysis

Two interfaces—*SimEventSource* for the source component and *SimEventListener* for the listener—are created to capture the different roles in the listener mechanism. It is good design to use interfaces to distinguish the dual roles that a component can hold. *SimEventSource*'s interface is implemented by an over-arching *BasicSimEventSource* abstract class while *SimEventlistener*'s interface is implemented by one of the most-derived *SimEntityBase* abstract class. Placing both implementations under a common abstract class would have facilitated analysis.

BasicSimEventSource is a new topmost generalized class in the entity class hierarchy. This may seem to suggest that this class has abstracted some fundamental functionality. However, the sequence diagram in Figure 37 shows a dependency of this topmost generalized abstract class on its derived abstract class. The *notifyListener* method of the *BasicSimEntitySource* contacts the *processSimEvent* method of *SimEntityBase*. This dependency is facilitated through the *SimEventListener* interface's *processSimEvent* method. Otherwise, there is inverse relationship between the abstract classes.

The mechanism for event triggering that propagates across each listener component uses the *processSimEvent* method. The mechanism for event triggering for the component itself uses the *handleSimEvent* method. The *handleSimEvent* method in fact uses the *processSimEvent* method. Event triggering for inter- and intra-component could have used one common interface, possibly rendering the *SimEventListener* interface redundant.

When a scheduled event occurs, separate steps of the event-triggering mechanism are carried out intra-componently (Figure 34) followed by inter-componently (Figure 37). The significant difference of these two steps lies in the entry point for each event execution, since each listener component is distinct from the source component. A means to consolidate and execute all these entry points together—perceiving all components alike—would have sufficed, achieved with one single step of the event-triggering mechanism.

D. EVENT MODELING LANGUAGE

UML is the de-facto modeling language in the software industry. This thesis work has used UML extensively in the study of design patterns, Java listener mechanisms, and Simkit. UML has a suite of powerful graphical representations that help analyze and express software-design artifacts and capture the design relationship and interactivity between classes, interfaces and objects.

Event-graph methodology is a set of graphical representations used to analyze and design a DES system from an event-driven perspective. It is a powerful graphical representation that helps analyze and design the interactivity of events. Event-driven perspective is an abstract form of system analysis that does not have a direct mapping to a class, interface, or object. UML diagrams of class, sequence, and activity could have been used to represent the interactivity of events. Activity diagrams would have been the best UML graphical representation to model event-graph logic. However event-graph methodology advocates the means to schedule as well as cancel events. A UML activity diagram cannot represent cancellations of events. This inadequacy, coupled with considerations in using a class or object or interface, would soon clutter the analysis process and eventually loose focus on an event-driven perspective.

In a Simkit simulation application, events are the central and active elements in the analysis and design of a DES. Simkit supports a direct transition from a system's event-graph design into its implementation as the modeling paradigm establishes a correspondence of an event to a method (containing the event-execution logic) residing in the entity's class.

A Simkit component corresponds to an object in UML; events (and their interactivity) of a Simkit component form the dynamics of the system, while objects (and their methods and interactivity) form the dynamics in UML. It is possible to use a UML interaction diagram (either a message-sequence or collaboration diagram) to model the interactivity of events, since method is the level of granularity that reflects the interactivity of events. Figure 38 shows a UML message-sequence diagram of the event-graph logic of a component. This is when UML is used to match the granularity of an event representation and the collaboration diagram is used to reflect the dynamics of the component.

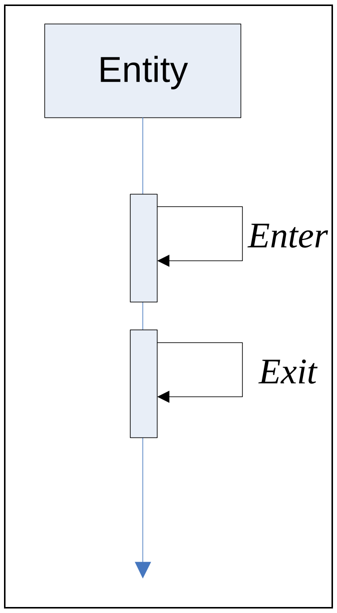


Figure 38 UML Message-Sequence Diagram of a Component

Figure 39 shows the corresponding event-graph representation of the same component. Figure 39 is able to presents the dynamics of the component better in terms of the possible avenues by which an event can be triggered. This is due to the difference in perspectives in the analysis of the dynamics of a component between UML collaboration diagrams and event-graph diagrams. Event-graph diagrams, which focus on the interactivity of events, provide succinct analysis on the dynamics of the system, making it the suitable event-modeling language for a DES.

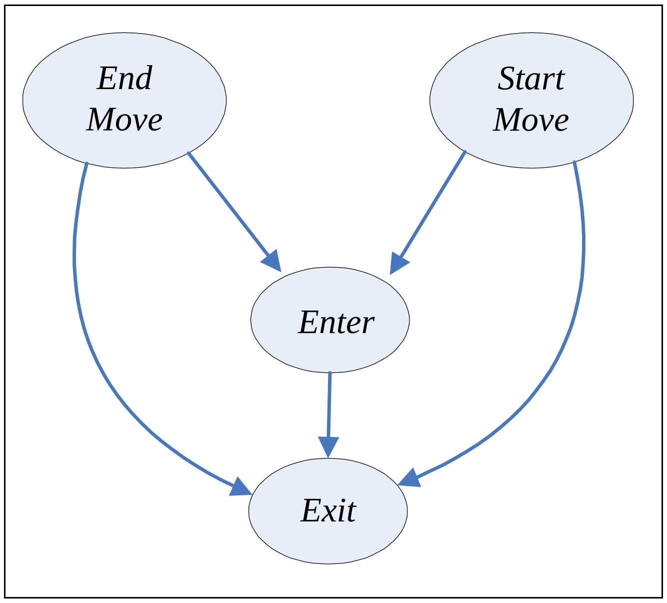


Figure 39 Event-Graph Diagram Of The Same Component

Both UML and event-graph are useful modeling languages. Understanding the strength of the language and the characteristics of the system under analysis lead to the right tool for the right job. This thesis work has substantial understanding of both modeling languages, and they are used complementarily. UML has been used to analyze and design the simulation engine, while the design of the mechanism is an implementation of event-graph methodology.

VII. PROPOSED DESIGN PATTERNS FOR SIMKIT

This chapter focuses on a new design for the random-utility package, the intra-component mechanism, and the inter-component mechanism of Discrete Event Simulation Kernel (DESK). Detailed analysis for each module is conducted. The design and implementation are illustrated for each module. For the random-utility package, a discussion on empirical analysis is conducted. Finally, this chapter looks how the FPPS application has been ported over to the DESK to prove the compatibility and completeness of the modeling paradigm the DESK has achieved.

A. RANDOM NUMBER

The random-number module is an important utility to most Simkit-based application. The understanding on the existing design and the analysis of its implementation has provided insights on possible new designs. This section looks at a proposed design for this module and discusses the implementation in this research work.

1. Analysis

In the random package of Simkit, several key roles and their behavior have been identified. Understanding the behaviors in the existing design of the random package and the study of design patterns reveal that some form of design patterns can be suitably applied into this utility package.

The *RandomVariateFactory* provides clients a common point of contact in their request to create a concrete random variate. There should be only one object of the *RandomVariateFactory* in the system as it manages the set of RandomVariates. The presence of a duplicate managerial object will be confusing to clients and, more detrimentally, cause inconsistency in the results of repetitive simulation runs. There is therefore a need to avoid accidental creation of this managerial object. In this aspect, the Singleton DP has behavioral characteristics that fit the needs of the *RandomVariateFactory* managerial object.

In applying the Singleton DP, a new random variate factory will be the Singleton class—*RVFactory* Singleton class. As a Singleton class, it will provide global access for all clients to the one-and-only managerial object, who manages a common set of random variates for the entire system. This is the only object that will accept requests from all clients to create RandomVariate. The *slt* method is a globally accessible method to all clients in the system. As the Singleton DP only allows the Singleton class itself to create an object of this class, any accidental creation of this managerial object is prevented. The design of applying the Singleton DP on *RVFactory* is illustrated in Figure 40.

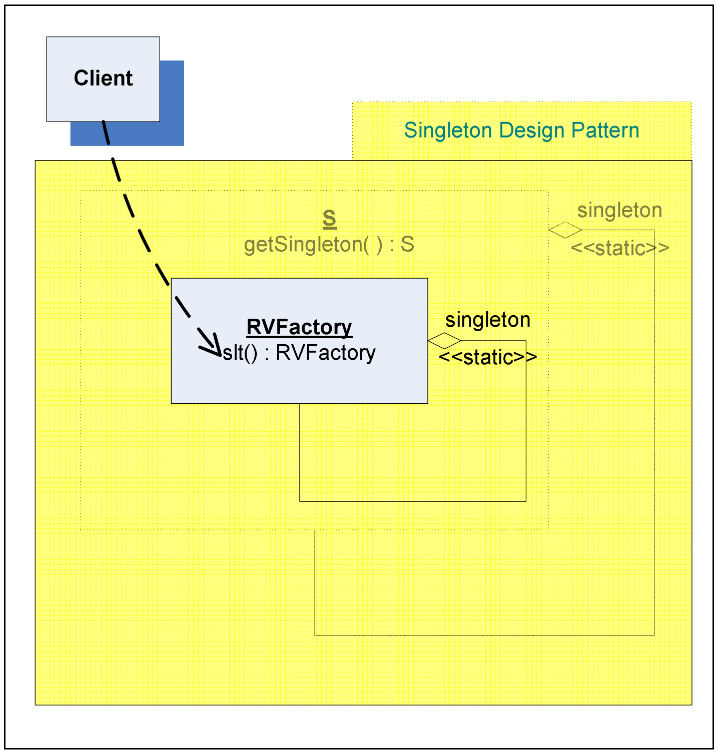


Figure 40 Applying the Singleton Design Pattern for RVFactory Class Diagram

The *RandomVariate* holds the key interface where different random-number generators and random variates will be implemented. The design of this interface plays the critical role in binding any future client to the existing *RandomVariateFactory* in its request for a random variate that might be written in the future.

When a client requests a random variate, the *RandomVariateFactory* has to discover and load the requested concrete random-variate class, create that object, and initialize it before it is made available to the client. Figure 31 has shown that there are several steps that are involved in creating a concrete random variate. The *RandomVariateFactory* has assumed the role of administering a common approach in creating each concrete random-variate object and keeping all the how-to hassle away from clients. The role that administers the common creation of RandomVariate also caters to any concrete random variates that will be written in the future, while the *RandomVariateFactory* has been built a priori. This analysis of the *RandomVariateFactory* from Simkit's random package shows behavioral characteristic where the factory DPs can be applied suitably in creating concrete random variates.

Applying factory design patterns, the concrete subclasses are the various concrete random-variate classes that will have the know-how of a new concrete random variate. The *RVFactory* provides the mechanism such that knowledge of which concrete random variate to create is encapsulated and kept out of the framework mechanism. The need to reveal the know-how is delayed till runtime, when the client makes the creation request and the *RVFactory* contacts the specific concrete random variate. The *RVFactory* is able to ensure that each RandomVariate is properly initialized before it is made available to the client. The design of classes where the factory DP is applied is illustrated in Figure 41.

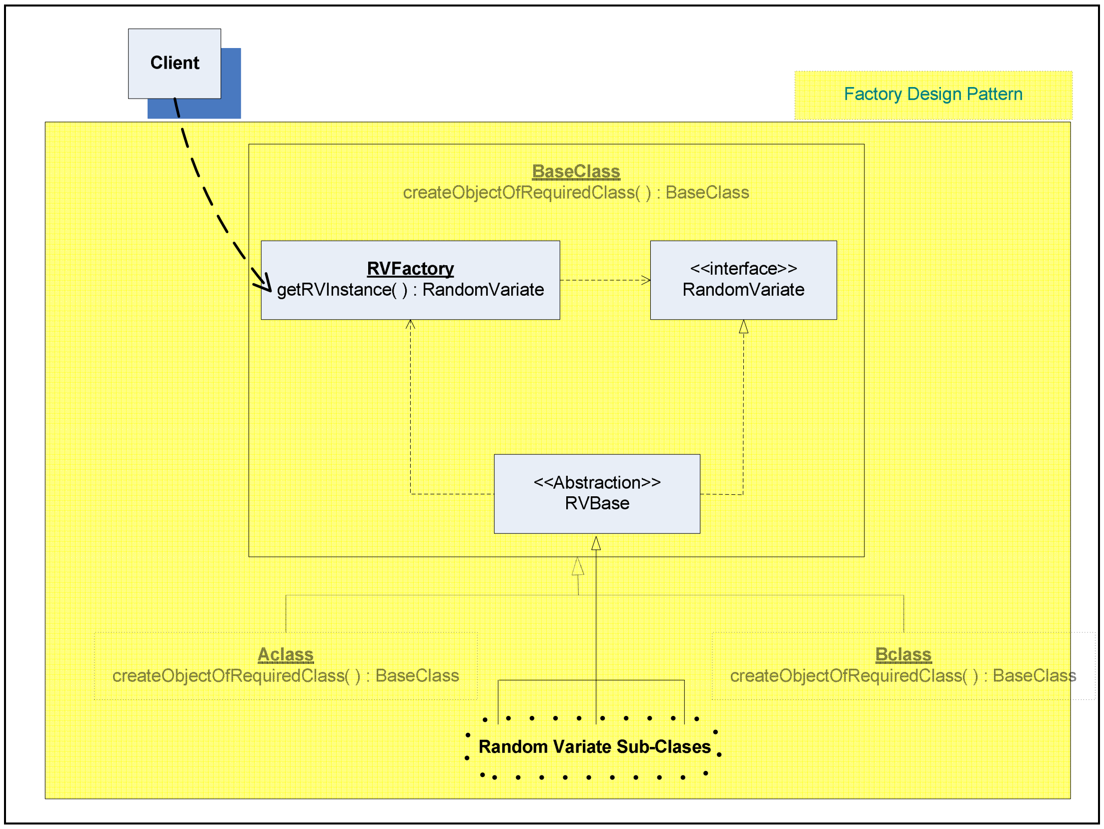

Client
Factory Design Pattern
BaseClass
createObjectOfRequiredClass( ) : BaseClass
RVFactory
getRVInstance( ) : RandomVariate
<<interface>>
RandomVariate
<<Abstraction>>
RVBase
Aclass
createObjectOfRequiredClass( ) : BaseClass
Bclass
createObjectOfRequiredClass( ) : BaseClass
Random Variate Sub-Clases

Figure 41 Applying Factory Design Pattern for Random Variate Class Diagram

2. Design and Implementation

The new design for the random package has incorporated two design patterns. The key classes, abstractions, and interfaces are shown in Figure 42. The *RVFactory* is the singleton where all clients will be contacting. It will administer the creation mechanism for concrete random variates and maintain the set for the system. The *slt* (Singleton publicly accessible method) is a request for the common managerial object. The *getRVInstance* (get random-variate instance) method is a request for the creation of a concrete random variate. This method is the interface that binds, yet decouples, the *RVFactory* that creates concrete random variate from the clients. Both these methods are named specifically to avoid ambiguity to the clients in contacting the *RVFactory*.

In creating a new random variate, the *RVFactory* needs to discover, load, create, and initialize that object. The first client in the system that requests the creation of a new random variate has to pay the cost of delayed time in discovering and loading the new random-variate class. This has led to the consideration of a pre-loading step by each concrete random variate when the system starts up. This pre-loading step will register the concrete random variate with the *RVFactory*. During runtime, when a request is made by any client, there is no time delay to discover and load the concrete random-variate class since it has been registered. This registration has established a dependency by the *RVBase* on the *RVFactory*.

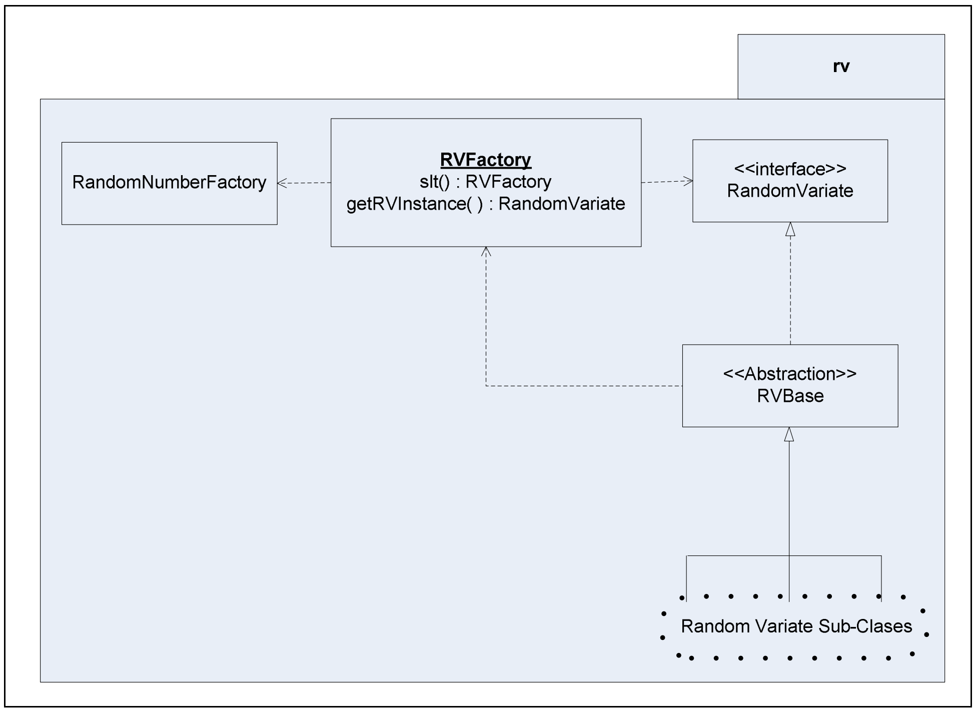


Figure 42 New Random-Package Top-Level Class Diagram

Figure 43 illustrates the interaction between the concrete random variate (*DerivedRV*), *RVBase,* and *RVFactory* in the pre-loading process. At system startup, as each concrete random variate registers with the *RVFactory* the class will be discovered and loaded into the system.

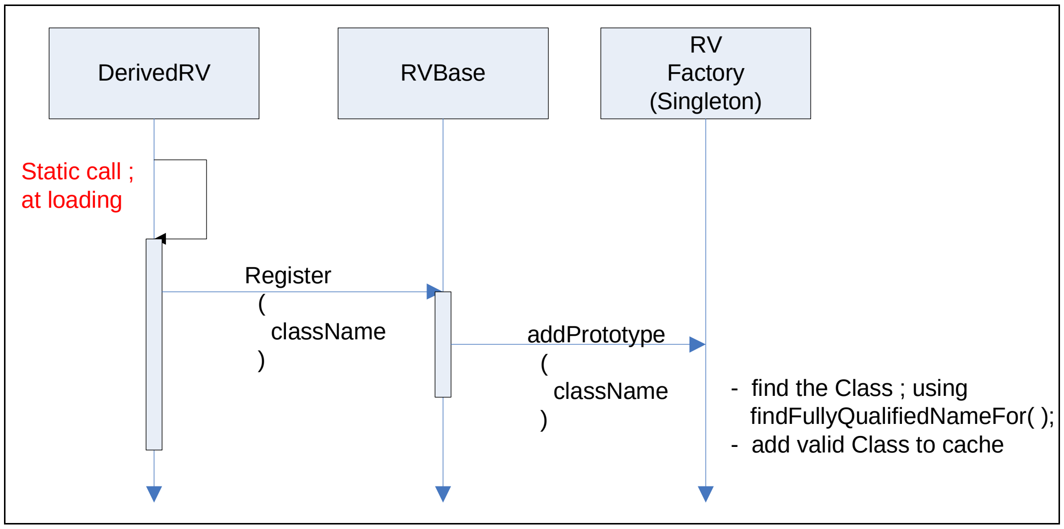

DerivedRV
RVBase
RV Factory (Singleton)
Static call ; at loading
Register (className)
addPrototype (className)
- find the Class ; using findFullyQualifiedNameFor( );
- add valid Class to cache

Figure 43 Pre-loading of Random Variate Sequence Diagram

During runtime, when client requests the creation of a random variate, the *RVFactory* simply instantiate an object from its cache and initializes it accordingly, as illustrated in Figure 44.

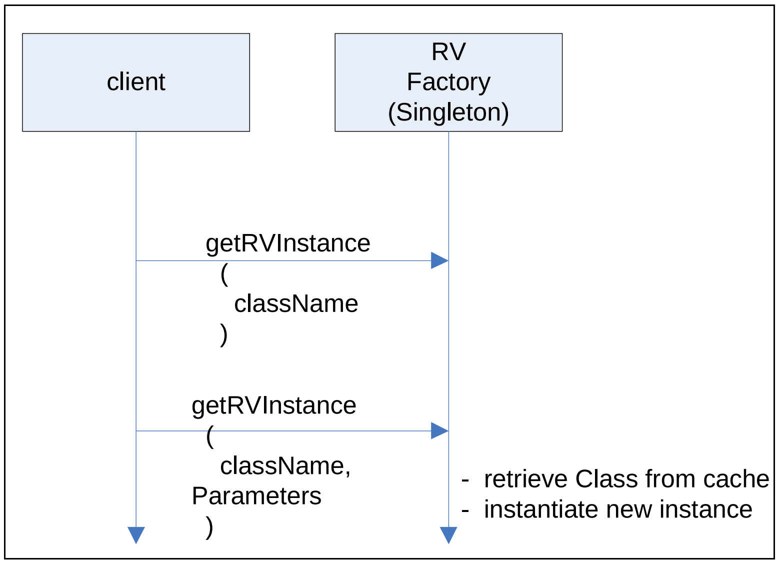

client
RV Factory (Singleton)
getRVInstance (className)
getRVInstance (className, Parameters)
- retrieve Class from cache
- instantiate new instance

Figure 44 Creating Random Variate Using New Random-Package Sequence Diagram

3. Empirical Analysis

In experiments, empirical analysis of the runtime performance of the proposed *rv* package, in comparison with the *simkit.random* package, was conducted.

In the tests, a total of twenty random variates that are subclasses of the *RandomVariate* abstract class have been created. One object instance of each the *random.RandomVariate* is requested through the *simkit.random* package. System performance is measured in terms of the process computation (CPU) time (in msec) consumed in creating these instances.

Similarly, a total of twenty random variates that are subclasses of the *RVBase* abstract class have been created. One object instance of each *rv.RandomVariate* is requested through the *rv* package. As the new design in *rv* package splits up the creation process into a pre-loading and a creation step, system performance is measured in terms of process computation time that is consumed for the pre-loading and creation steps for these instances.

Number of RandomVariate instances	***simkit.random*** **CPU time (msec)**		***rv*** **CPU time (msec)**	
	Creation		Pre-loading	Creation
1	15		15	0
3	16		15	0
5	16		16	0
10	31		16	0
15	46		20	0
20	52		31	0

Table 1 Empirical Results of *simkit.random* Package vs. *rv* Package

The results of empirical tests have shown that applying a design pattern in the *rv* package yields a gain in system performance on the pre-loading step. At runtime, all random objects that are requested, including the first object, incurred insignificant computation time.

B. INTRA-COMPONENT EVENT SCHEDULING

The mechanism of the simulation engine that facilitates intra-component event scheduling makes it possible for components to be independent. This section looks at the proposed design of a new mechanism that attempts to inject elements of elegance and simplicity without compromising component independence. This section examines the design of the new mechanism and discusses its implementation in this research work.

1. Analysis

An understanding of Simkit and the modeling paradigm that Simkit advocates has identified the important role the simulation engine assumes in providing unanimous mechanisms to schedule and trigger events. This role facilitates the application modeling work's focus on the problem domain. The modeling paradigm where the entity is an independent component that houses all event-execution logic is an elegant concept within a discrete-event-simulation framework.

A close study of the event-triggering mechanism in Figure 34 has revealed that when a scheduled event occurs, the event-execution entry point residing in the entity (*Application Derived Sim Entity*) must be resolved before execution can be triggered to occur. The triggering of such an entry point is in fact a generic step of method invocation. One idea in designing DESK is to abstract and encase this generic step into a single atomic object that solely invokes the method. The challenge in this idea is finding the means to resolve the entry point of the entity when the event is scheduled.

This challenge led to the concept of a method callback: encapsulation of a method invocation that is the entry point of event execution of an entity. A method-callback object executes method invocation as an atomic transaction. With a method-callback object, method invocation is now independently decoupled away from the entity itself.

An event is still the important component from the event-driven perspective in modeling a DES system. The new design in DESK will attempt to elevate the emphasis on event in this modeling paradigm. Due consideration is given to entrusting the event to assume an active role in the event-triggering mechanism. This is viable when the concept of method callback is associated with event. When a scheduled event occurred, the associated method callback will directly trigger the event-execution entry point on the entity. This association eliminates the dependency of event on entity. An occurring event now assumes an active role that encompasses the entire event-triggering mechanism. The entity now assumes a more passive role of hosting the system-state variables and business-logic behavior. The new design concept in DESK involving method callback has defined a finer level of granularity in atomic element of execution, as compared to that of an entity. This new design of an active event and its associated method callback is in fact a behavioral characteristic where the Observer DP has been applied. The event—the Subject— actively notifies its method callback—the Observer— when the scheduled time of occurrence has arrived—the update call. The design where the method callback encases the method invocation is a neat approach of decoupling the Subject away from the Observer.

A scheduler will be designed as the main contact point for scheduling and ordering events in order and assuming the role of retrieving scheduled events to occur.

2. Design and Implementation

The classes of the new design in DESK are illustrated in Figure 45. There are two packages: *eb* (the event bus) package and the *ent* (the entity) package. The *eb* package contains the *Event* class. An *Event* is associated with a *MCB* (method-callback). The *Scheduler* is the Singleton that is associated with multiple *Event*s. The *ent* package contains the *EntityBase* abstract class. This class defines the mechanism of event scheduling for *Application Derived Sim Entity*. The *ent* package is dependent on the *eb* package.

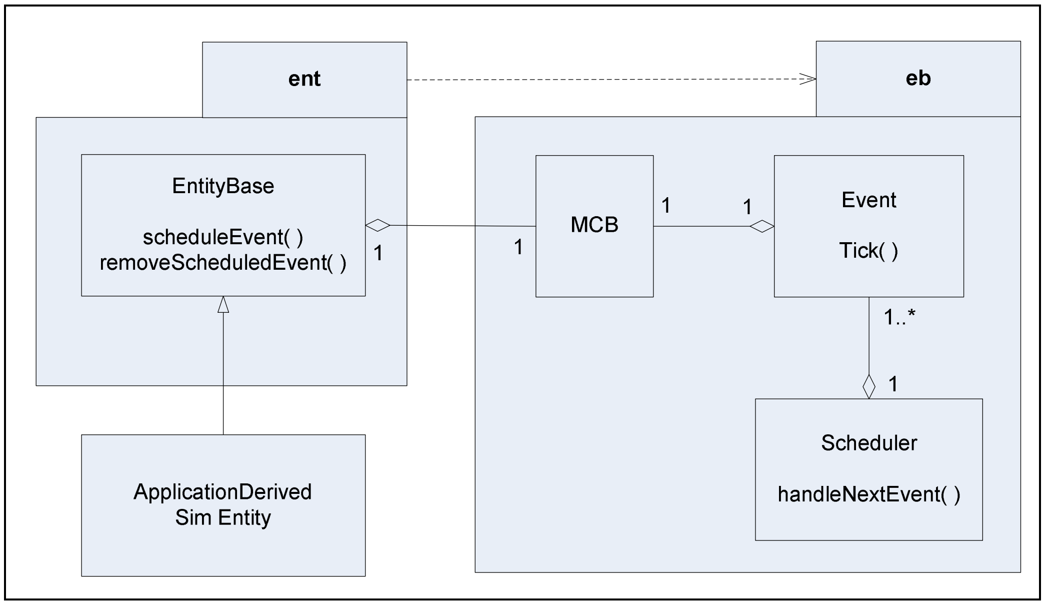


Figure 45 DESK Event Scheduling Class Diagram

Figure 46 illustrates the interaction of the entity of the *Application Derived Sim Entity* with the new classes of DESK in scheduling an event. The entity uses the *scheduleEvent* method of the *EntityBase* to schedule an event according to its business logic. The *MCB* corresponding to the scheduled event is retrieved by the *EntityBase*, which is associated to the newly created *Event* object. *Scheduler* is contacted to insert the scheduled event. The corresponding cancellation of scheduled event is in Appendix B.

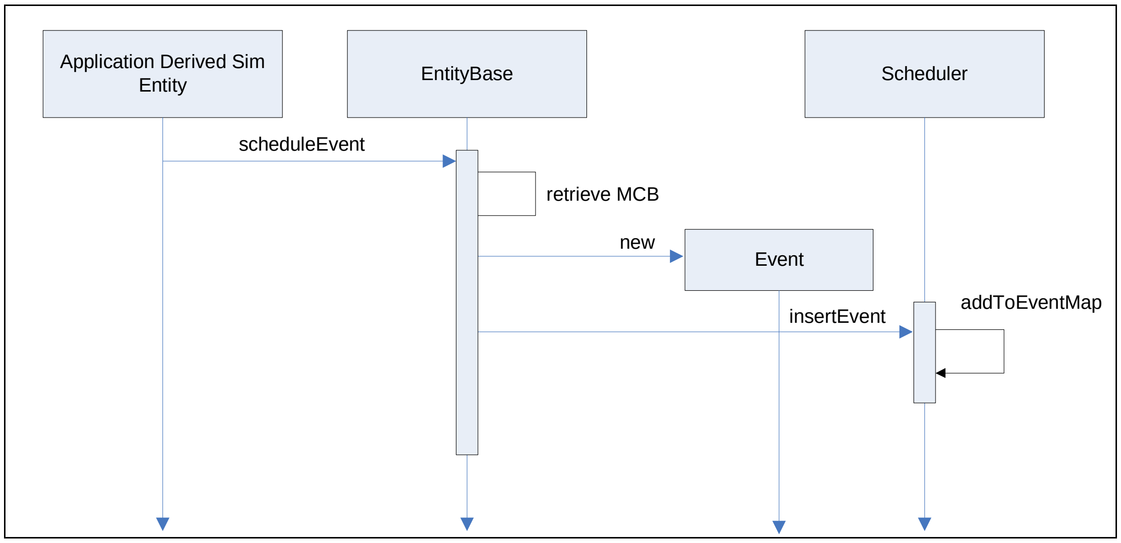


Figure 46 DESK Event Scheduling Sequence Diagram

Figure 47 illustrates the interaction among classes of DESK in handling the event-triggering mechanism when a scheduled event occurs. After the event is retrieved by the *Scheduler*, the associated *MCB* will trigger the entry point residing on the *Entity* directly. It illustrates the active role of the event in this new design.

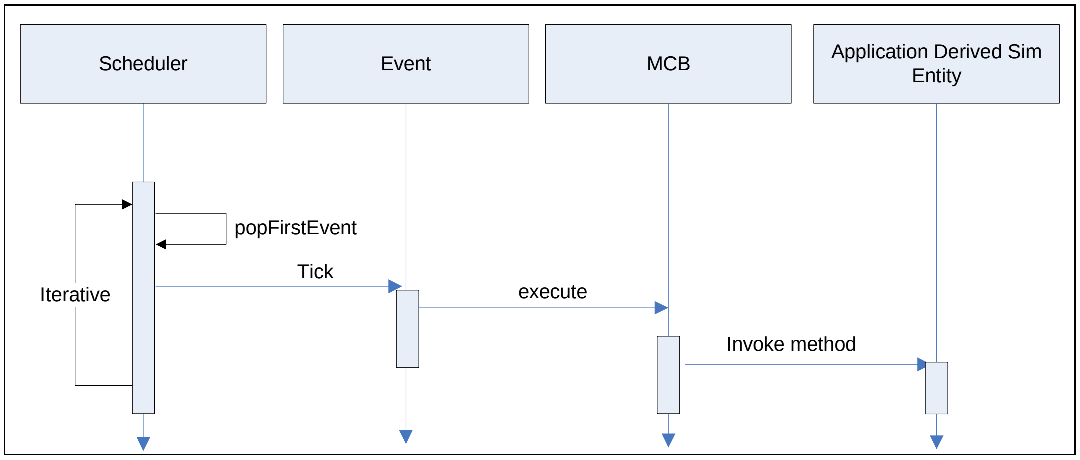


Figure 47 DESK Event Triggering Sequence Diagram

C. INTER-COMPONENT EVENT SCHEDULING

The mechanism of the simulation engine that facilitates inter-component event scheduling makes it possible to assemble components to build larger complex systems. This section looks at a proposed design where the mechanism for intra-component event scheduling can be easily extended to support inter-component event scheduling. The proposed design of the new mechanism reflects simplicity, extensibility, and maturity in the framework. This section will look at the design of this new mechanism and discuss its implementation in this research work

1. Analysis

The LEGO [11] framework advocates the assembling of ready-made components (as is), propagating the event-triggering behavior of a component across other components through the component-listener mechanism, to form chained interactivity. This conceptual approach of building larger complex system is an excellent way for software components to be truly plug-and-play in component-based-simulation modeling.

The new design in DESK also attempted to incorporate the concepts of LEGO [11] framework. One key consideration is that an entity needs to assume a dual role as source component and as listener. There is no limitation on the number of listening components that can be associated with a source component. One challenge is to devise a common event-triggering mechanism for both intra- and inter-component. Just as the LEGO [11] concept of building larger complex system is to use a component as is, the underlying mechanism in incorporating LEGO concepts into listener mechanisms should also use the event-triggering mechanism component as is.

A close study on the design of classes for DESK in Figure 45 shows that the method-callback object is decoupled away from the entity. Figure 47 shows that event triggering is carried out by the method callback as an atomic transaction. Propagating the event triggering to a component listener essentially requires the listener component's method-callback to execute. The design is easily extended, such that an event—the Subject—is associated with many method callbacks—the Observers. When a scheduled event occurs, all the method-callbacks—source and listener components alike—are executed. The new design in DESK simply uses the same as-is event-triggering mechanism to support the listener mechanism of the LEGO framework. This design also shows that applying the Observer DP for the basic mechanism results in a simplistic design that supports ease of extensibility. The design where the Observer DP has been incorporated is illustrated in Figure 48.

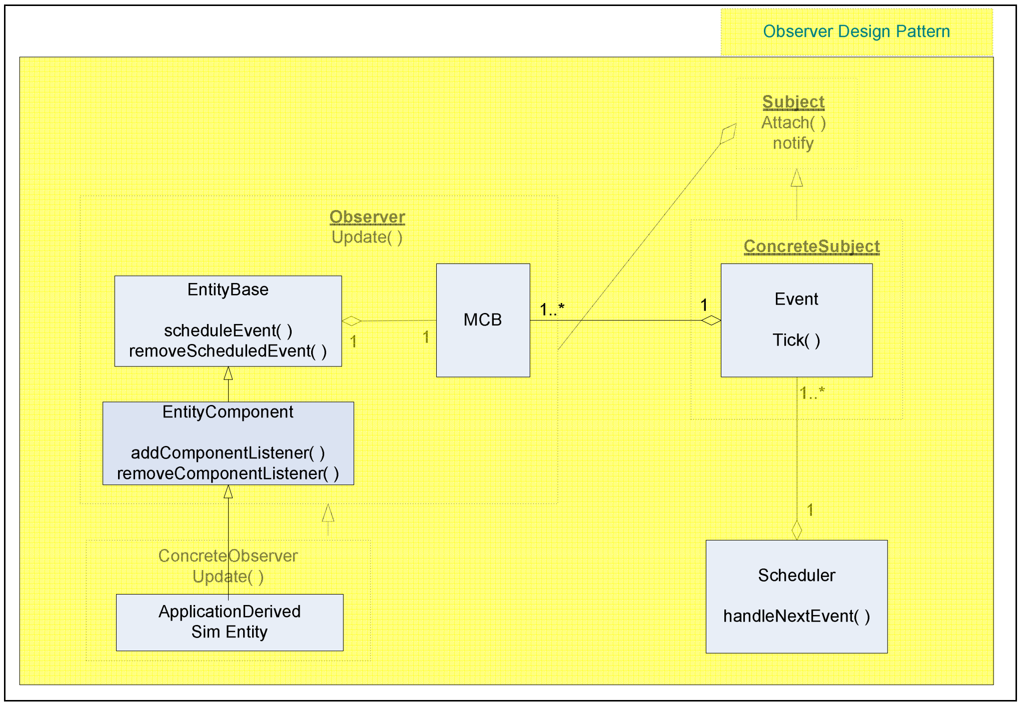


Figure 48 Applying Observer DP for Intra- And Inter-Component Class Diagram

2. Design and Implementation

The new abstract class—*EntityComponent*—is illustrated in Figure 49. This class contains the mechanism to support establishment of the linkage-source component and its listener. Listeners are added into the source component through the *addComponentListener* method. The *Event* is now associated to many *MCB*, where each *MCB* belongs to each component. Figure 49 shows that the existing design of DESK for intra-component event scheduling is easily extended to support inter-component event scheduling.

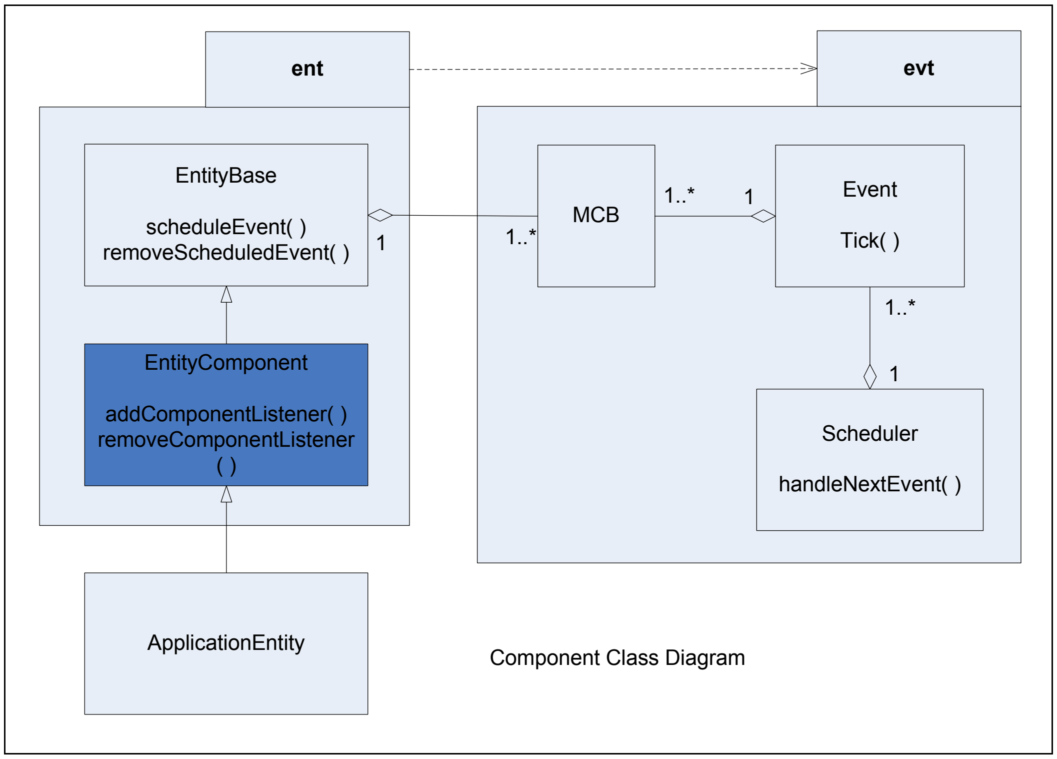


Figure 49 DESK Component Listener Class Diagram

Figure 50 illustrates the interactivity between *Application Derived Sim Entity* and *EntityComponent* in adding a component listener. *EntityComponent* will need to retrieve the *MCB* from the component listener. This abstract class consolidates the *MCB* for each event that the source component is capable of scheduling. When an event is scheduled by the source component, the consolidated *MCB*s are associated with the event that it creates. The corresponding removal of listener component is in Appendix C.

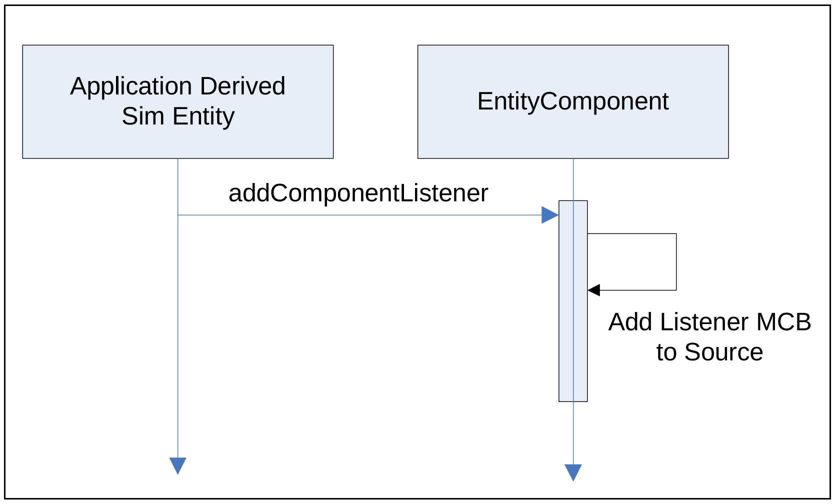


Figure 50 DESK Add-Component Sequence Diagram

Figure 51 illustrates that when a scheduled event occurs, the single event triggering mechanism is carried out. Each *MCB* that is associated with the *Event* is triggered—source component and listener components alike.

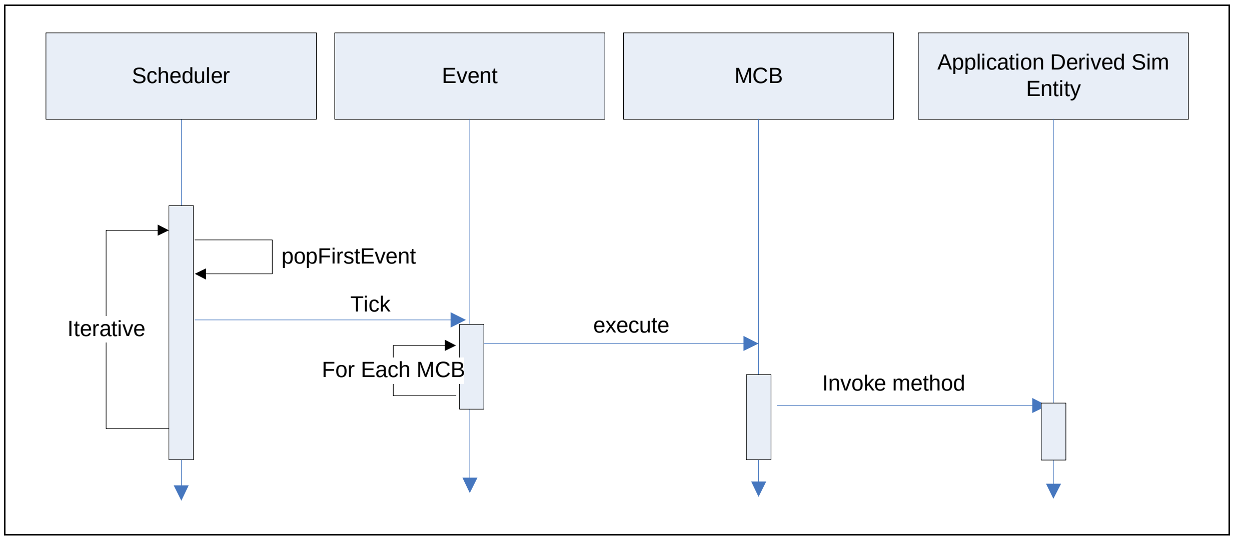


Figure 51 DESK Triggering Component Sequence Diagram

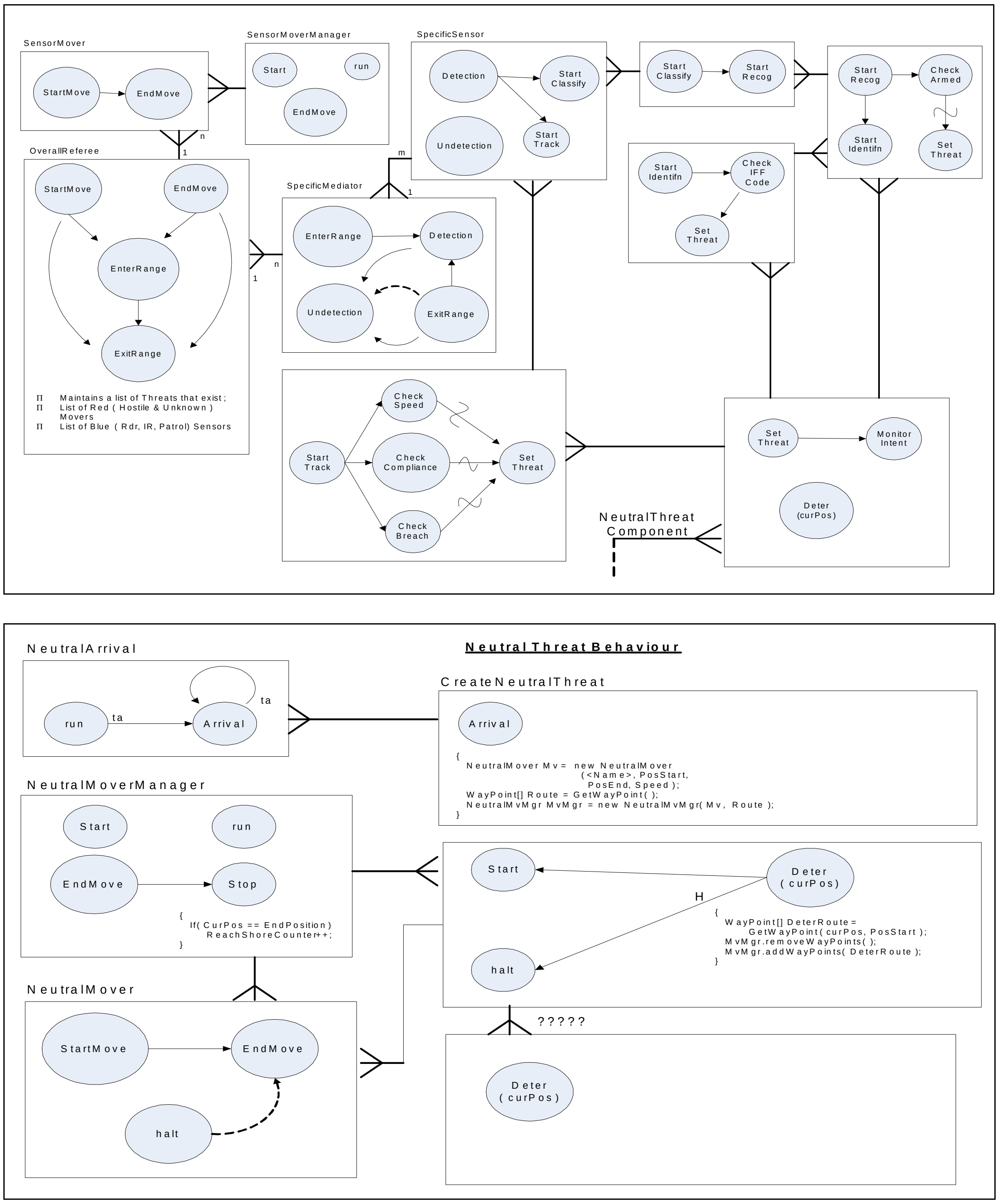
The design of the EntityComponent abstract class has also encompassed the concept of adapter as described in Chapter II, C. Simkit. The implementation is illustrated in Appendix C.

D. DES SIMULATION APPLICATION

The new design of DESK has implemented some features with compatible functionalities in Simkit. The *eb* and *ent* packages contain the implementation for the modeling paradigm and listener mechanism. An approach to test out the compatibility of the implementation in DESK is to use a DES simulation application that was developed using Simkit to run on DESK without changing any business logic of the application. The Force Protection and Port Security (FPPS) simulation that the author co-developed for the Systems Engineering and Analysis Project 11 (SEA-11) and delivered in June 2007 is a good application to test for testing the compatibility of the DESK, in terms of the completeness of the compatible functionalities that DESK has implemented.

The FPPS simulation system attempts to address several issues faced by the Port of Oakland as regards unknown and impromptu threats. It is a busy port, and its vulnerability has a detrimental effect on the economy, both at the national and international levels. Much of the effort undertaken consisted of analysis of different kinds of threats and an assessment of current assets and levels of readiness, proposing several alternatives to improve the readiness of the port and offering cost-effectiveness analysis of different solutions. The system identified that terrorist threats would come by sea.

Figure 52 shows the event-graph that captures the business logic of the FPPS simulation application.



```
{
   NeutralMover Mv =  new NeutralMover
                        (<Name>, PosStart,
                         PosEnd, Speed );
   WayPoint[] Route = GetWayPoint( );
   NeutralMvMgr MvMgr = new NeutralMvMgr( Mv, Route );
}
```

```
{
   If( CurPos == EndPosition)
      ReachShoreCounter++;
}
```

```
{
   WayPoint[] DeterRoute =
        GetWayPoint( curPos, PosStart );
   MvMgr.removeWayPoints( );
   MvMgr.addWayPoints( DeterRoute );
}
```

Figure 52 Event-graph Logic of the FPPS Simulation Application

To port a Simkit application over to run on DESK, all components whose class subclass from *SimEntityBase* of the *simkit* package only needs to switch over to sub-class from *EntityComponent* of the *ent* package from DESK. All implemented business logic of the application will run seamlessly. Figure 53 shows the FPPS application where the same business logic runs seamlessly after it has been successfully ported over to run on DESK.

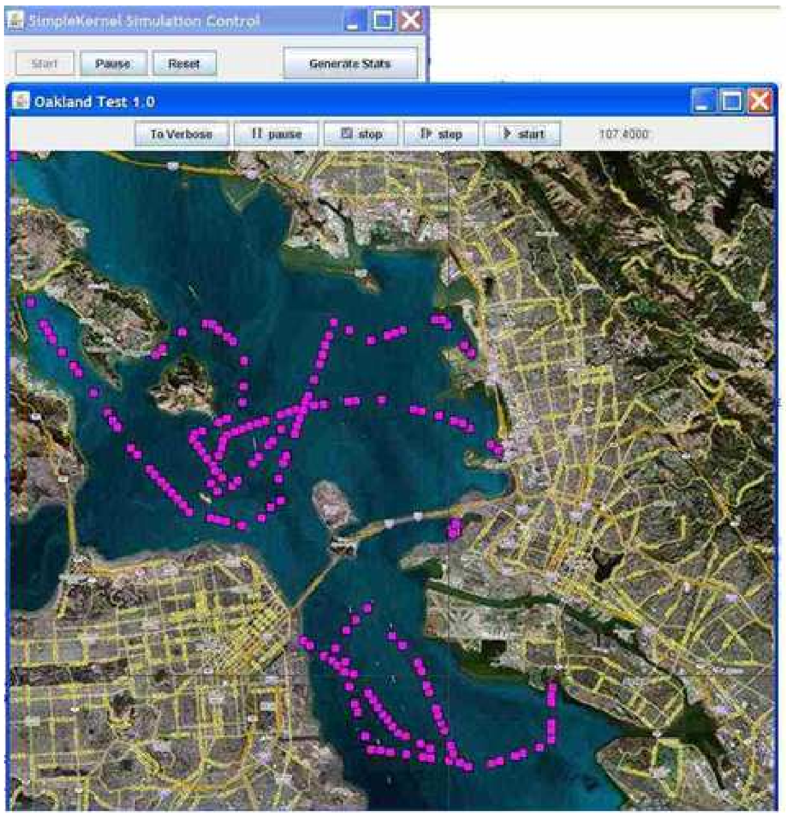


Figure 53 FPPS Simulation Application Running on DESK

VIII. CONCLUSION

The research work in this thesis has devoted substantial effort to studying and acquiring an adequate understanding of the relevant methodology, modeling language, and state-of-the-art technology of DES. A detailed understanding of design patterns has helped identify the behavioral characteristics that exist inherently in Java framework and Simkit simulation engine. These have similar concepts, but the underlying behavior showed different design patterns at play. This has established DP relevance and reinforced the fact that design patterns are well-devised solutions that evolve over time.

In this research, the Simkit random utility was studied. The existing designed was re-examined and a new design proposed. The new design—*rv* package—incorporated relevant design patterns whose behavioral characteristics have been suitably identified. Empirical testing was conducted to compare and contrast the robustness of both the existing and new design. The performance gained of the new design showed how this research work has met its objective of incorporating relevant design patterns and tackled its challenge of averting the conflict between elegance and performance.

This work also studied the underlying mechanisms of the Simkit simulation-engine framework. The features of the modeling paradigm were reviewed in totality and the existing mechanism designed re-examined. A design— DESK (*eb* package and *ent* package) —was proposed, representing a new approach that incorporates a suitably identified design pattern in the inter-workings of the mechanisms through the concept of method callback. A simpler design was created. The inter-workings of mechanisms are more extensible, flexible, and maintainable. The new design showed how this research work has pursued the motivating challenge of elegance in mechanism design within the simulation-engine framework. The simplicity and elegance of the new design that supports the modeling paradigm would certainly elevate the maturity of the simulation-engine infrastructure. An existing simulation application—FPPS—from the SEA-11 project was used to demonstrate the compatibility of the new design.

This research may be termed a success. The author has gained much insight on the modeling paradigm from the framework infrastructure perspective. Grasp of this work apparently not only facilitates an understanding of the simulation infrastructure for simulation-application developers, but also fosters an appreciation of the simulation infrastructure for the “simulationist” working on the internals of the simulation engine. The satisfaction the simulationist feels in working within an infrastructure that advocates rapid development of a DES system amounts to a joy and fulfillment that is beyond thrilling.

There are many avenues for future research work based on this thesis. One direction is other design patterns—identifying their relevance and applicability and proposing how they may be introduced into the simulation-engine framework. Simkit has a suite of modules, both core and utility packages. Another great opportunity for follow-on work would be re-investigating some of these modules and proposing how new design can add elegance, maturity, and robustness to the simulation engine.

APPENDIX A. RANDOM NUMBERS

A. RANDOM VARIATE CLASSES

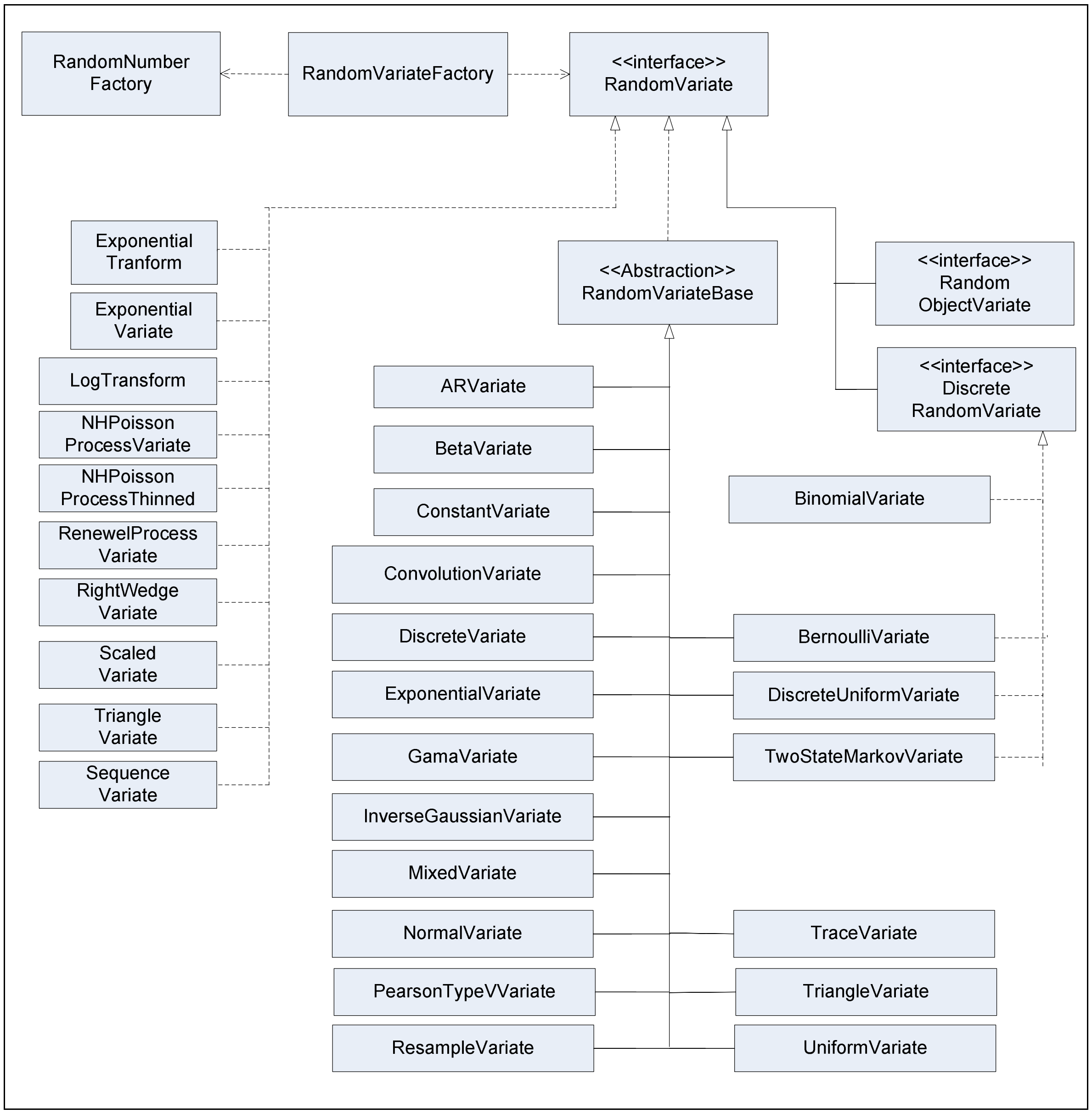


Figure 54 Existing Simkit.Random package, All Random-Variate Classes

B. RANDOM NUMBER CLASSES

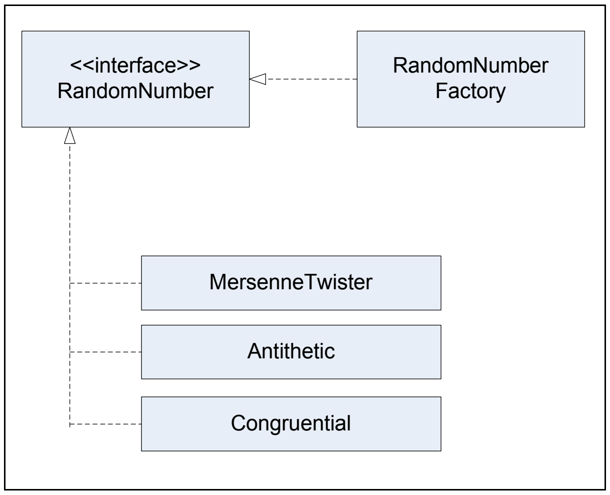


Figure 55 Existing Simkit.Random Package, Random Numbers

C. INTERFACES

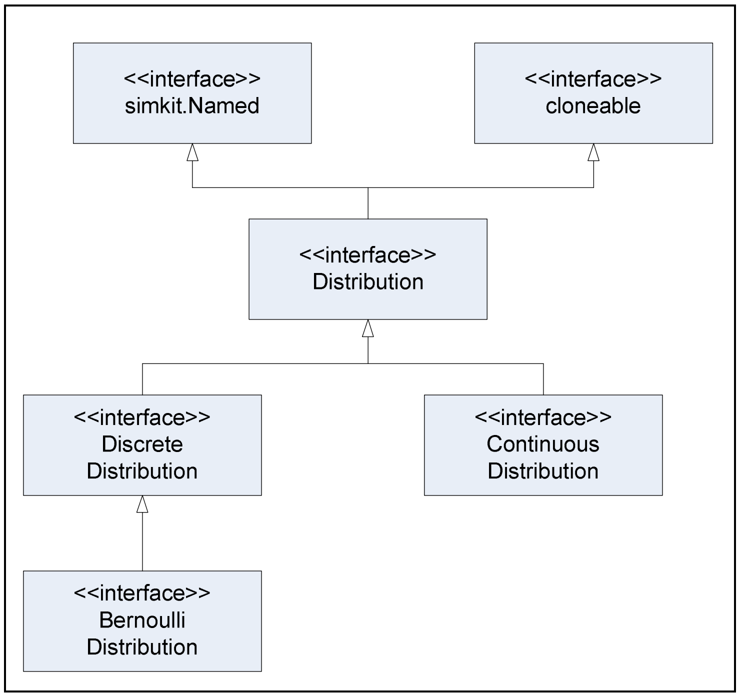


Figure 56 Existing Simkit.Random Package, All Interfaces

D. APPLICATION CODE SAMPLE

```
// Existing Design
// Using simkit.random package
//  runtime request for a random variate

simkit.random.RandomVariate Random01Variate =
            RandomVariateFactory.getInstance( “var.Random01” );
```

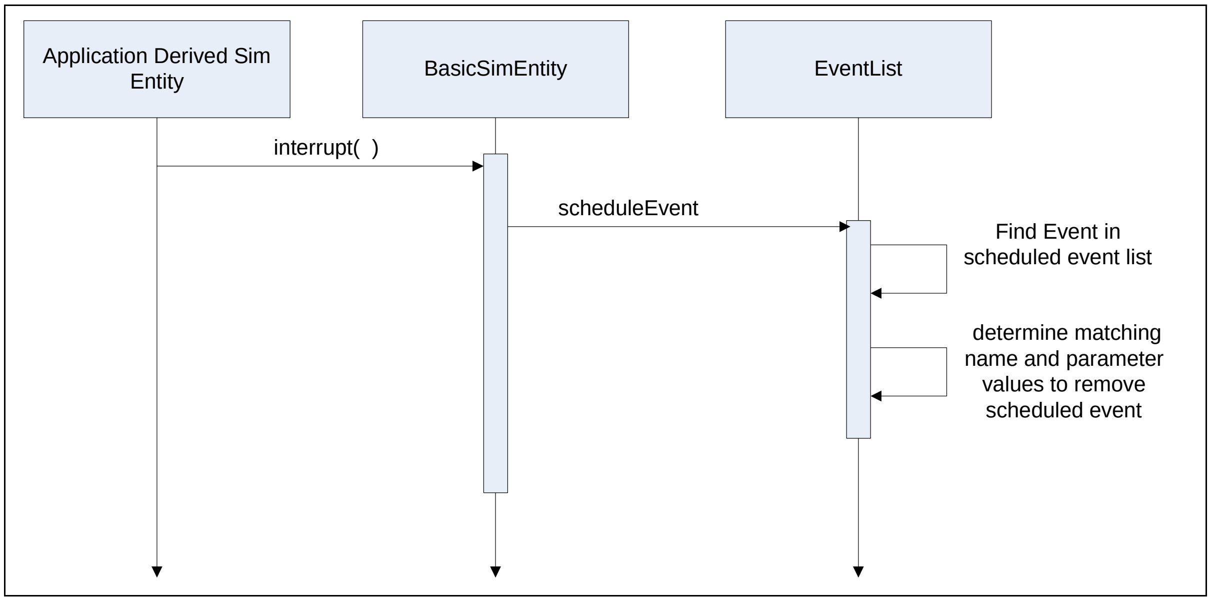
```
// New Design
// Using rv package
//
// Pre-loading step
RVFactory.slt().preLoad( "var.RV01Derived");


// runtime request for a random variate
rv.RandomVariate RV01 = RVFactory.slt().getRVInstance("RV01Derived" );
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. INTRA-COMPONENT

A. SIMKIT EVENT CANCELLATION



Application Derived Sim Entity
BasicSimEntity
EventList
interrupt( )
scheduleEvent
Find Event in scheduled event list
determine matching name and parameter values to remove scheduled event

Figure 57 Existing Simkit, Cancellation of Scheduled-Event Sequence Diagram

B. DESK ENTITY INITIALIZATION

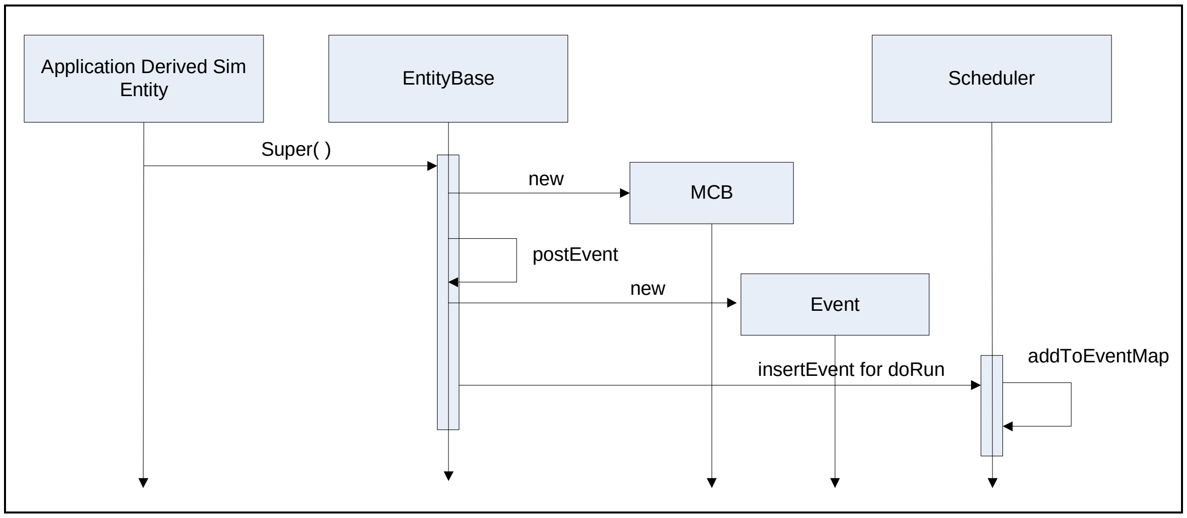


Figure 58 New DESK, Entity-Initialization-Routine Sequence Diagram

C. DESK EVENT CANCELLATION

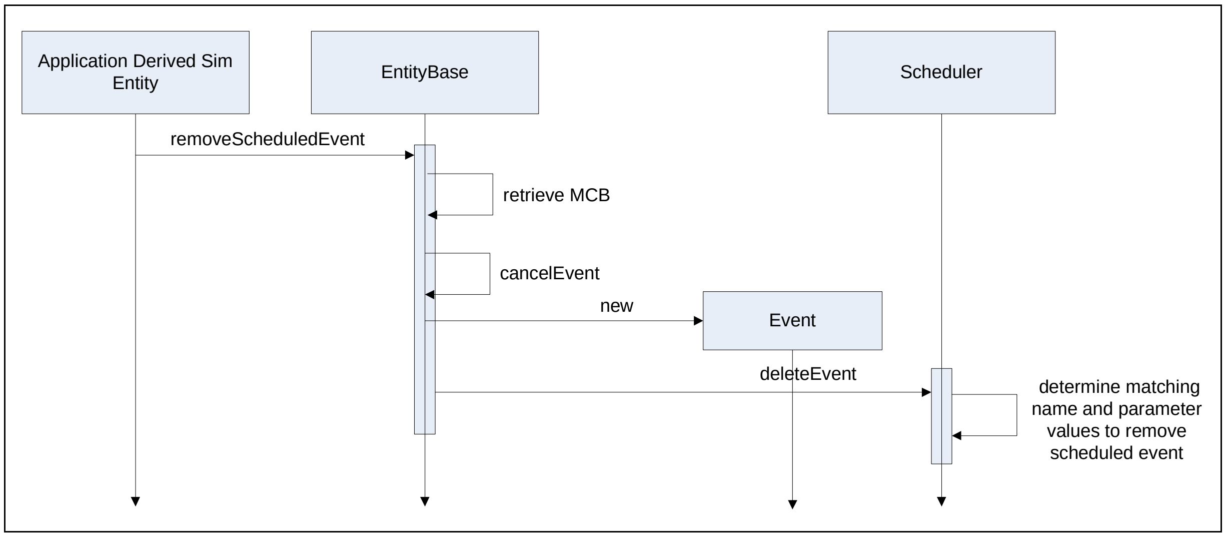

Application Derived Sim Entity
EntityBase
Scheduler
removeScheduledEvent
retrieve MCB
cancelEvent
new
Event
deleteEvent
determine matching name and parameter values to remove scheduled event

Figure 59 New DESK, Cancellation-of–Scheduled-Event Sequence Diagram

```
public boolean removeScheduledEvent( String eventName, Object... parameters )
  {
     boolean removeScheduleEventStatus = false;
     List<CallBack> callBackList = null;
    String eventNameSignatureString = createMethodSignatureString( eventName, parameters );
      if( methodCBMap.containsKey( eventNameSignatureString ) )
     {
        callBackList = methodCBMap.get( eventNameSignatureString );
        removeScheduleEventStatus = cancelEvent( eventNameSignatureString, callBackList, parameters );
     }
     return removeScheduleEventStatus;
  }
protected static boolean cancelEvent( String eventName, List<CallBack> cbList, Object... parameters )
  {
     boolean cancelEventStatus = false;
     DiscreteEvent cancellingEvent = new DiscreteEvent(
           eventName, 0.0, parameters );
     cancellingEvent.setEventSubscribers( cbList );
     cancelEventStatus = Scheduler.slt().deleteEvent( cancellingEvent );
     return cancelEventStatus;
  }
```

Figure 60 New DESK, Cancellation-of–Scheduled-Event Code Snipplet

APPENDIX C. INTER-COMPONENTS

A. SIMKIT REMOVE-EVENT LISTENER

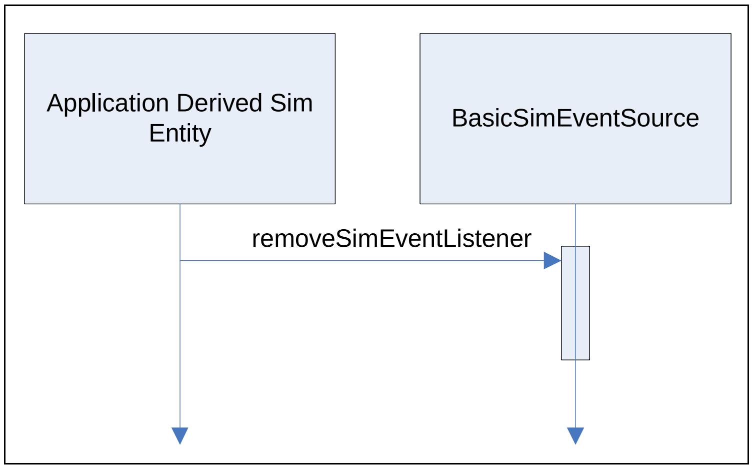


Figure 61 Existing Simkit, Remove-Listener-Component Sequence Diagram

B. DESK REMOVE-EVENT LISTENER

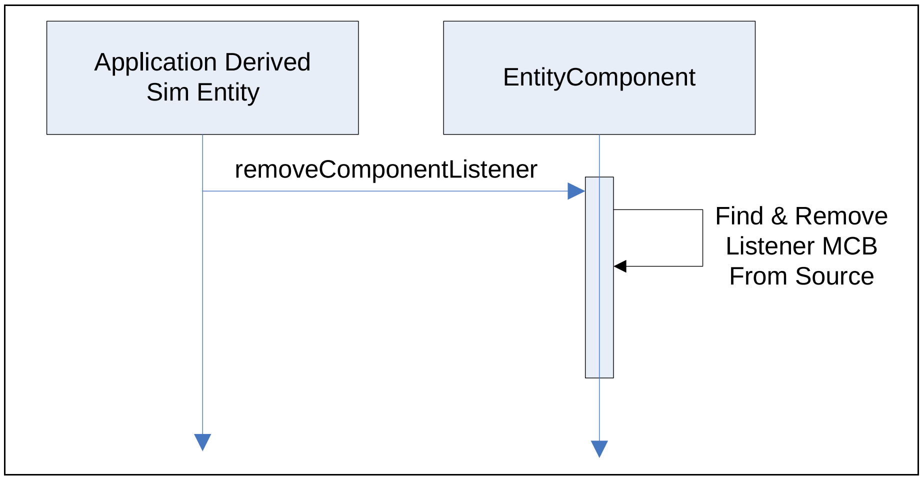

Application Derived Sim Entity
EntityComponent
removeComponentListener
Find & Remove Listener MCB From Source

Figure 62 New DESK, Remove-Listener-Component Sequence Diagram

```
public boolean removeComponentListener( EntityBase listener )
{
    boolean removeListenerStatus = false;
    Set<String> keys = methodCBMap.keySet();
    for( String curEventName  : keys )
    {
        if( inExcludeEventList( curEventName ) )
            continue;
        List<CallBack> curCBList = methodCBMap.get(curEventName);
        CallBack thisCB = curCBList.get( 0 );
        CallBack listenerCB = listener.getCallBack( curEventName );
        if( thisCB.isCompatible( listenerCB ) )
        {
            if( curCBList.contains( listenerCB ))
            {
                removeListenerStatus =
                    curCBList.remove( listenerCB );
            }

        }
    }
    return removeListenerStatus;
}
```

Figure 63 New DESK, Remove-Listener Component Code Snipplet

C. DESK ADD ADAPTER

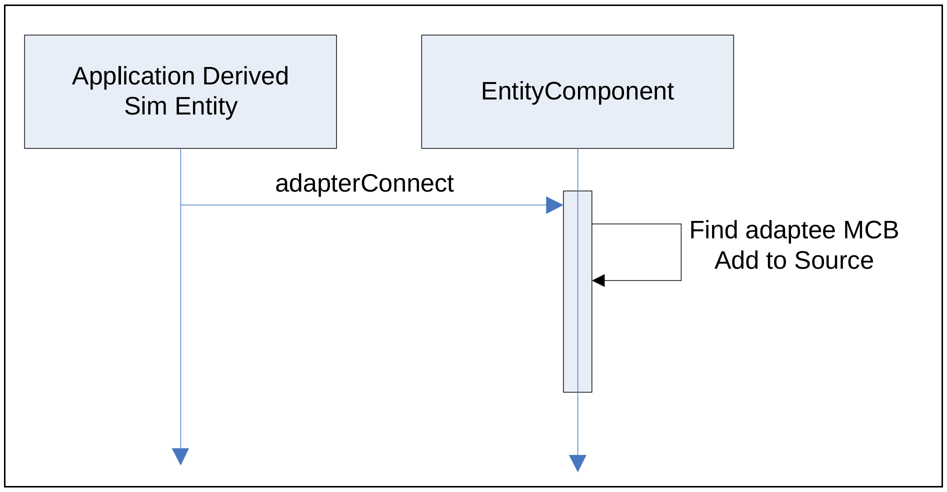


Figure 64 New DESK, Connect-an-Adaptee-Component Sequence Diagram

D. DESK REMOVE ADAPTER

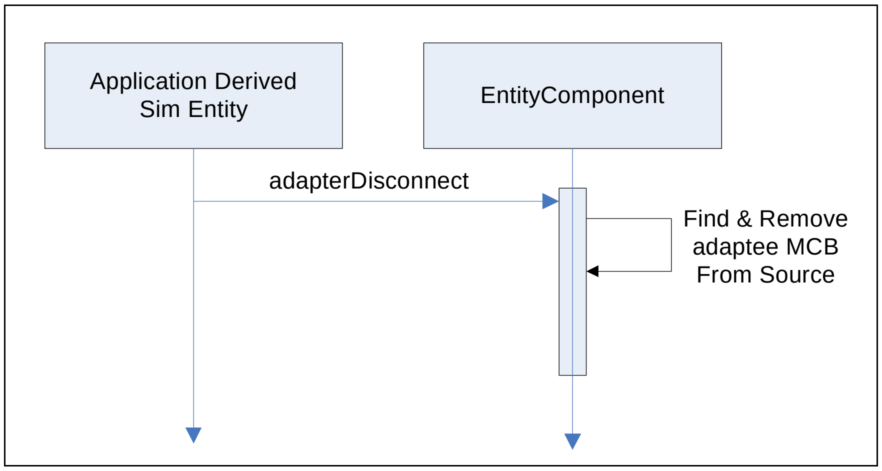

Application Derived Sim Entity
EntityComponent
adapterDisconnect
Find & Remove adaptee MCB From Source

Figure 65 New DESK, Remove-an-Adaptee-Component Sequence Diagram

```
public boolean adapterConnect( String sourceEvent, String adapteeEvent,
        EntityComponent adaptee, Object... params )
{
    boolean connectionStatus = false;
    String adapteeEventSignatureString = createMethodSignatureString( adapteeEvent, params );
    CallBack adapteeCB = adaptee.getCallBack( adapteeEventSignatureString );
    if( adapteeCB != null )
    {
        List<CallBack> CBList = findCBList( sourceEvent, params );
        if( CBList != null )
        {
            if( !( CBList.contains( adapteeCB ) ) )
            {
                connectionStatus = CBList.add( adapteeCB );
            }
        }
    }
    return connectionStatus;
}

public boolean adapterDisconnect( String sourceEvent, String adapteeEvent, EntityComponent adaptee, Object... params )
{
    boolean disconnectionStatus = false;
    String adapteeEventSignatureString = createMethodSignatureString( adapteeEvent, params );
    CallBack adapteeCB = adaptee.getCallBack( adapteeEventSignatureString );
    if( adapteeCB != null )
    {
      List<CallBack> CBList = findCBList( sourceEvent, params );
      if( CBList != null )
      {
          if( CBList.contains( adapteeCB ) )
          {
              disconnectionStatus = CBList.remove( adapteeCB );
          }
      }
    }
    return disconnectionStatus;
}
```

Figure 66 New DESK, Adapter Code Snippet

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995, ISBN 0-201-63361-2.

[2] A. M. Law, *Simulation Modeling And Analysis, 4th Edition*, McGraw Hill, 2007, ISBN-10: 0-07-298843-6.

[3] A. H. Buss, "Modeling with Event Graphs," in *Proceedings of the 1996 Winter Simulation Conference*, 1996, pp. 153-160.

[4] L. Schruben (Cornell University), "Simulation Modeling with Event Graphs," *Communications of the ACM,* vol. 26 Number 11, pp. 957-963, November 1983.

[5] J. P. Hayes, *Computer Architecture and Organization (Second Edition)***.** McGraw Hill Publishing, 1998, ISBN 0-07-027366-9.

[6] E. Lee, and T. Park, "Dataflow Process Networks," in *Proceedings of IEEE*, 1995, vol. 83, pp. 773-799.

[7] K. D. Tocher, *The Art of Simulation.* D. Van Nostrand Co., Inc., Princeton, New Jersey, p. 147.

[8] T. G. Poole, and J. Z. Szymankiewicz, "Using Simulation to Solve Problems," *The Journal of the Operational Research Society*, vol. 29, No. 6, pp. 618-619 , June 1978.

[9] J. L. Peterson, *Petri Net Theory and the Modeling of Systems.* Prentice Hall, ISBN 0-13-661983-5.

[10] A. H. Buss, "Component Based Simulation Modeling," in *Proceedings of the 2000 Winter Simulation Conference*, 2000, pp. 964-971.

[11] A. H. Buss and P. J. Sánchez., "Building complex models with LEGOs (listener event graph objects)," in *Proceedings of the 2002 Winter Simulation Conference,* 2002, pp. 732-737.

[12] A. H. Buss, "Component Based Simulation Modeling with Simkit," in *Proceedings of the 2002 Winter Simulation Conference*, 2002, pp. 243-249.

[13] S. Bran, "What's New in UML 2.0," IBM Rationale Software, July 2007, ftp://ftp.software.ibm.com/software/rational/web/whitepapers/intro2uml2.pdf, last accessed August 2007.

[14] Object Management Group, “Unified Modeling Language: Infrastructure Version 2.0,” December 2003 Publication, http://www.uml.org/#UML2.0, http://www.omg.org/docs/ptc/03-09-15.pdf, last accessed July 2007.

[15] Object Management Group, “Unified Modeling Language: Superstructure Version 2.0,” August 2005 Publication, http://www.uml.org/#UML2.0, http://www.omg.org/docs/formal/05-07-04.pdf, last accessed July 2007.

[16] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual* (2nd edition). Addison-Wesley, 2005.

[17] Object Management Group, “Introduction to OMG’s UML,” July 2005, http://www.omg.org/gettingstarted/what_is_uml.htm, last accessed July 2007.

[18] M. Randy, “Practical UML: A Hands-on Introduction for Developers” , Borland Software Corporation, December 2003, http://dn.codegear.com/article/31863, last accessed September 2007.

[19] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson and I. F. King and S. Angel, *A Pattern Language,* Oxford University Press, New York, 1977.

[20] E. Braude, *Software Design: From Programming to Architecture.* John Wiley & Sons, Inc., Boston University, 2004, ISBN: 0-471-42920-1.

[21] Sun Microsystems, “Java Platform Standard Edition 6.0 API Specification,” May 2007, http://java.sun.com/javase/6/docs/api/, last accessed June 2007.

[22] Sun Microsystems, “Java Tutorials: JavaBeans,” August 2007, http://java.sun.com/docs/books/tutorial/javabeans/, last accessed August 2007.

[23] A. H. Buss and D. K. Ahner, “Dynamic Allocation of Fires and Sensors (DAFS): A Low-resolution Simulation for Rapid Modeling,” in *Proceedings of the 2006 Winter Simulation Conference,* 2006, pp. 1357-1364.

[24] A. H. Buss and K. A. Stork, “Discrete Event Simulation on the World Wide Web using Java,” in *Proceedings of the 1996 Winter Simulation Conference,* 1996, pp. 780-785.

[25] A. H. Buss and L. Jackson, “Distributed Simulation Modeling: A Comparison of HLA, Corba and RMI,” in *Proceedings of the 1998 Winter Simulation Conference,* 1998, pp. 819-825.

[26] A. H. Buss and P. J. Sánchez, “Simple Movement and Detection in Discrete Event Simulation,” in *Proceedings of the 2005 Winter Simulation Conference,* 2005, pp. 992-1000.

[27] A. H. Buss, “Basic Event Graphs Modeling,” *Simulation News Europe, Technical Notes,* Issue 31, April 2001.

[28] D.C. Matthew, “An exploratory analysis of waterfront Force Protection measures using simulation,” Master’s Thesis, Naval Postgraduate School, Monterey, CA, 2002.

[29] J. S. Patrick, “Evaluating the effectiveness of waterside security alternatives for Force Protection of navy ships and installations using X3D graphics and agent-base simulation,” Master’s Thesis, Naval Postgraduate School, Monterey, CA, 2006.

[30] K. Arnold, J. Gosling and D. Holmes, *The Java^{TM} Programming Language Fourth Edition (The Java^{TM} Series).* Addison Wesley, 2006.

[31] M. Naftalin, and P. Wadler, *Java Generics and Collections.* O’Reilly, 2007, ISBN-10: 0-596-52775-6.

[32] Sun Microsystems, “Java Tutorials: Writing Event Listeners,” August 2007, http://java.sun.com/docs/books/tutorial/uiswing/events/index.html, last accessed August 2007.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Arnold H. Buss
   Naval Postgraduate School
   Monterey, California

4. Man-Tak Shing
   Naval Postgraduate School
   Monterey, California

5. Singapore Technologies Electronics (Training And Simulation System) Pte Ltd
   Singapore